Universidade Federal de Minas Gerais - UFMG

Manual do Usuário do Ambiente de Simulação ProVANT

Autor: Arthur Viana Lara
Versão: 1.0

2 de agosto de 2018

Resumo

O objetivo deste manual é descrever os procedimentos necessários para a utilização do simulador ProVANT. Nele é apresentado desde o processo de instalação até a realização de testes de estratégias de controle via simulação. O Texto está organizado como a seguir:

1. O primeiro capítulo apresenta o contexto no qual esse ambiente de simulação está inserido e também uma breve introdução à VANTs.

2. O segundo capítulo apresenta os passos necessários para instalação do ambiente de simulação ProVANT;

3. O terceiro capítulo descreve o fluxo de utilização do ambiente de simulação, detalhando todas as funcionalidades da interface gráfica, tais como a escolha do cenário, modelo, estrategia de controle e aeronave;

4. O quarto capítulo é o mais importante para o usuário. Ele descreve os procedimentos para a implementação de novas estratégias de controle no ambiente de simulação.

5. O Apêndice A explica a estrutura de arquivos por trás dos modelos dos VANTs utilizados. Nesse capítulo são apresentadas as principais informações sobre modelagem cinemática e a importação de arquivos do software CAD para a plataforma Gazebo;

6. O Apêndice B descreve os plugins utilizados no ambiente de simulação.

7. Os Apêndices C e D descrevem os arquivos CMakelists.txt e package.xml.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Veículos aéreos não tripulados (VANTs) são aeronaves equipadas com sistemas embarcados, sensores e atuadores que permitem a realização de voos autônomos ou remotamente controlados. Eles são comumente classificados em dois grupos: veículos de asas rotativas, como helicópteros e quadrotores, e veículos de asas fixas, como aviões.

Há diversas aplicações para VANTs, alguns exemplos são:

- Pulverização de culturas;
- Condução de rebanhos;
- Monitoramento de estradas;
- Inspeção da linhas de energia;
- Entrega de suprimentos em locais de difícil acesso.

O simulador apresentado neste manual está associado ao ProVANT[1]. O ProVANT consiste em uma parceria entre a Universidade Federal de Santa Catarina e a Universidade Federal de Minas Gerais, com o objetivo de realizar pesquisas e desenvolver novas tecnologias para aperfeiçoar o desempenho de VANTs. Neste contexto, atualmente, o ProVANT está focado no desenvolvimento de VANTs Tilt-rotor. O Tilt-rotor é uma aeronave que possui configuração híbrida, portanto apresenta as principais vantagens das aeronaves de asa fixa e de asa rotativa, como por exemplo consumo reduzido de energia em voos de cruzeiro e decolagem e pouso na vertical. Ele pode operar tanto em ambientes fechados quanto abertos.

Atualmente o ProVANT possui três tipos de linhas de projeto:

1. Mecânico/Aerodinâmico;
2. Instrumentação/Eletrônica;
3. Projeto de estratégias de controle e estimação de estados;

O projeto Mecânico/Aerodinâmico já conta com 5 versões de VANTs que foram nomeadas como VANT 1.0, 2.0, 2.1, 3.0 e 4.0, as Figuras 1.1, 1.2, 1.3, 1.4 e 1.5 ilustram, respectivamente, essas versões. A Instrumentação/Eletrônica está em estágio avançados, com todos os circuitos eletrônicos desenvolvidos e realizando melhorias nos mesmos. Com relação ao Projeto de estratégias de controle e estimação de estados, no contexto do projeto diversas estratégias de controle foram propostas por alunos de mestrado e doutorado.

Ainda com relação ao Projeto de estratégias de controle e estimação de estados, alguns trabalhos científicos foram desenvolvidos. Em Donadel (2015) controladores baseados nas técnicas *Linear Quadratic Regulator* (LQR), $\mathcal{H}_\infty$ linear e $\mathcal{H}_2/\mathcal{H}_\infty$ linear misto foram desenvolvidos para o VANT 1.0. Com o objetivo de rastreamento de trajetória. Alguns destes controladores foram validados através de voos experimentais. Em de Almeida Neto (2014) é apresentado uma técnica de controle não-linear baseado em *feedback linearization* com o objetivo de transporte de carga para o VANT 2.0, ainda com o mesmo objetivo, Alfaro (2016) apresenta o desenvolvimento de um controlador baseado na técnica *Model Preditive Control* (MPC). Em Rego (2016), trabalhou-se com o transporte de carga do VANT 2.0, no entanto, abordou-se o problema de seguimento de trajetória do ponto de

[1] provant.eng.ufmg.br

vista da carga, para o qual foram projetados estimadores de estados robustos. Em Cardoso (2016), desenvolveu-se uma estratégia de controle adaptativo com a finalidade de seguimento de trajetória para o VANT 3.0.

Figura 1.1: Projeto mecânico VANT 1.0.

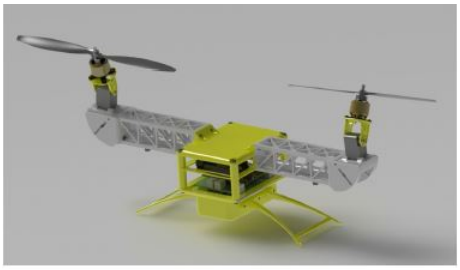

Figura 1.2: Projeto mecânico VANT 2.0.

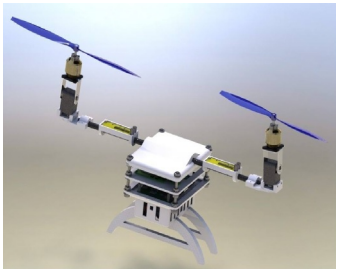

Figura 1.3: Projeto mecânico VANT 2.1.

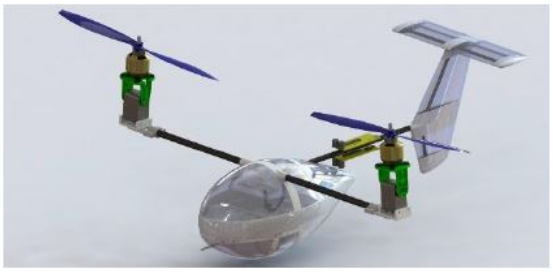

Figura 1.4: Projeto mecânico VANT 3.0.

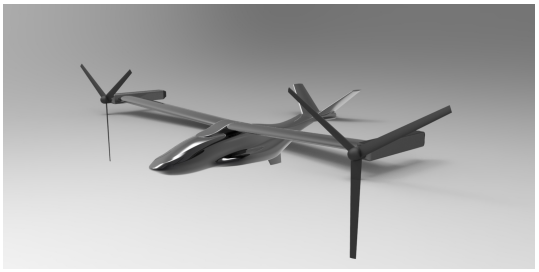

Figura 1.5: Projeto mecânico VANT 4.0.

O simulador ProVANT tem o objetivo de ser uma ferramenta confiável e de fácil utilização. O intuito é possibilitar a redução de custos e de tempo necessário para o projeto e validação de estratégias de controle.

Capítulo 2

Instalação

O simulador ProVANT é um ambiente de simulação desenvolvido com o objetivo de validar e avaliar o desempenho de estratégias de controle. A versão do simulador a qual esse manual se refere foi desenvolvida sobre as plataformas Gazebo 7 e o *framework* de desenvolvimento de aplicações robóticas *Robot Operating System* (ROS), na versão Kinect. Para utilizá-la é necessário um computador com **sistema operacional Ubuntu 16.04**

O ROS fornece uma interface de programação para aplicações em robótica e dispõe de repositórios com vários módulos de software. Já o Gazebo é um software de simulação 3D e de licença gratuita, atualmente sob a responsabilidade da Open Source Robotics Foundation (OSRF). Ele é capaz de simular o comportamento dinâmico de corpos rígidos articulados, e inclui funcionalidades como detecção de colisão e visualização gráfica.

Este capítulo apresenta os passos necessários para instalação do simulador ProVANT. Inicialmente, são detalhados os processos de instalação de plataformas utilizadas pelo simulador, como a biblioteca QT versão 5, os pacotes de software Git e ROS. Por fim, é apresentado o processo de instalação do ambiente de simulação ProVANT.

2.1 Procedimentos para instalação do simulador

Os procedimentos especificados a seguir assumem como premissa que o computador, o qual será instalado o simulador, está rodando uma versão limpa/recém-instalada do Ubuntu versão 16.04.

2.1.1 Instalando Git

O código fonte do simulador ProVANT está hospedado no Github. Para o acesso a esses arquivos é necessário primeiro que o usuário tenha instalado no computador o pacote Git. Caso não tenha, para instalá-lo, abra um novo terminal e execute os seguintes comandos:

```
$ sudo apt-get update
$ sudo apt-get install git
```

2.1.2 Instalando e configurando o ROS

O ROS fornece uma interface de programação para aplicações de robótica e vários módulos de software, entre eles o simulador Gazebo na versão 7, utilizado pelo simulador ProVANT. Esta subseção detalha as instruções necessárias para a instalação da distribuição ROS Kinetic Kame. A seguir estão detalhados os passos para a instalação do ROS[1].

[1] para mais detalhes sobre o processo de instalação, assim como problemas na instalação pode-se acessar a homepage do ROS, wiki.ros.org

Configure os repositórios do Ubuntu

Antes de iniciar a instalação é necessário configurar os repositórios do Ubuntu para níveis de permissão "restricted", "universe", e "multiverse.". Atenção: normalmente, após a instalação do Ubuntu 16.04, estas opções já se encontram configuradas. Caso não estejam, para obter instruções sobre como fazer isso siga o passo a passo apresentado em:

https://help.ubuntu.com/community/Repositories/Ubuntu.

Configure o sources.list

Também é necessário configurar o computador para aceitar o software de packages.ros.org. Para isso, abra um novo terminal e digite o seguinte comando:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main"
 >/etc/apt/sources.list.d/ros-latest.list'
```

Informe as chaves de criptografia para acesso aos repositórios do ROS

Isso pode ser realizado executando o seguinte comando:

```
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key
421C365BD9FF1F717815A3895523BAEEB01FA116
```

Instalação

Antes de iniciar o processo de instalação do ROS, é necessário atualizar o índice do pacote Debian, para isso execute:

```
$ sudo apt-get update
```

Tendo atualizado os pacotes Debian, pode-se fazer download dos arquivos binários do ROS

```
$ sudo apt-get install ros-kinetic-desktop-full
```

Inicialize o Rosdep

Antes de poder usar o ROS, é necessário inicializar o sistema Rosdep. O sistema Rosdep permite instalar as dependências do sistema, para o código fonte que deseja-se compilar. Ele é necessário para executar alguns dos componentes principais no ROS.

```
$ sudo rosdep init
$ rosdep update
```

Configuração de variáveis de ambiente

Para que as variáveis de ambiente do ROS sejam automaticamente adicionadas sempre que um novo terminal é iniciado, para isso execute os seguintes comandos:

```
$ echo "source /opt/ros/kinetic/setup.bash" >> $HOME/.bashrc
$ source $HOME/.bashrc
```

Crie um espaço de trabalho ROS

Por fim, é necessário criar um espaço de trabalho ROS, para isso execute a seguinte sequência de comandos no terminal:

```
$ mkdir -p $HOME/catkin_ws/src
$ cd $HOME/catkin_ws/
$ catkin_make
$ source $HOME/catkin_ws/devel/setup.bash
$ echo "source $HOME/catkin_ws/devel/setup.bash" >> $HOME/.bashrc
```

2.1.3 Instalando a biblioteca QT

Para a instalação apropriada da interface gráfica de configuração do ambiente de simulação ProVANT, é necessária a instalação da *Integrated Development Environment* (IDE) Qt Creator 5. Para isso, execute os seguintes comandos no terminal:

```
$ cd $HOME/Downloads
$ wget http://download.qt.io/official_releases/online_installers/qt-unified-linux-x64-online.run
$ sudo chmod +x qt-unified-linux-x64-online.run
$ ./qt-unified-linux-x64-online.run
```

2.1.4 Download e instalação do ambiente de simulação

O código fonte do ambiente de simulação ProVANT, está localizado no repositório:

https://github.com/Guiraffo/ProVANT-Simulator

Este repositório possui acesso particular, então, antes de prosseguir como processo de instalação, solicite acesso ao Prof. Guilherme Vianna Raffo[2].

Com o acesso liberado, para fazer o download da versão atual do ambiente de simulação ProVANT, basta digitar no terminal os seguintes comandos:

```
$ cd $HOME/catkin_ws/src
$ git clone https://github.com/Guiraffo/ProVANT-Simulator.git
```

Tendo realizado o download pode-se realizar a instalação e configuração do ambiente de simulação através da execução dos seguintes comandos:

```
$ cd $HOME/catkin_ws/src/ProVANT-Simulator
$ sudo chmod +x install.sh
$ ./install.sh
```

Caso todos os procedimentos descritos anteriormente tenham sido realizados com sucesso, o usuário estará pronto para prosseguir para o próximo capítulo, referente ao fluxo de utilização do ambiente de simulação ProVANT.

[2]E-mail para contato: raffo@ufmg.br

Capítulo 3

Fluxo de utilização

Este capítulo apresenta as funcionalidades disponíveis para a operação do simulador através da interface gráfica, tais quais:

- Seleção de cenário
- Seleção de modelo de VANT
- Configuração de cenário
- Edição de configurações do VANT
- Inicialização de simulação.

3.1 Como inicializar o ambiente de simulação

Tendo realizado o processo de instalação do simulador com exito, como descrito no capítulo anterior, pode-se iniciar o processo de desenvolvimento de controladores no ambiente de simulação ProVANT. Para iniciar a interface gráfica do usuário abra um novo terminal e digite o seguinte comando:

```
$ provant_gui
```

A interface gráfica de acesso ao simulador será então incializada e aparecerá a janela principal, como ilustrado na Figura 3.1.

3.2 Seleção de cenário

Tendo inicializado a interface gráfica do ambiente de simulação, é necessário que o usuário selecione um cenário de simulação. Isso pode ser realizado de duas maneiras diferentes, ambas através da utilização da opção "Simulation", presente no menu superior da interface gráfica, ilustrado na figura E.2:

(i) Ao clicar em "New", é possível abrir um cenário template com extensão ".tpl".

Um cenário template é um uma configuração de cenário que serve de molde para a criação de outros cenários. Ele não pode ser editado, necessita da adição de um modelo de um VANT e deve ser salvo antes da simulação com o formato ".world"

(ii) Ao clicar em "Open", é possível abrir um cenário já existente (arquivo com extensão ".world").

(iii) Ao clicar em "About", é possível conferir informações gerais do simulador ProVANT .

Ainda no menu "Simulation", é possível salvar um cenário com outro nome no formato ".world", clicando em ("Save"), e também finalizar o ambiente de simulação, clicando em ("Exit").

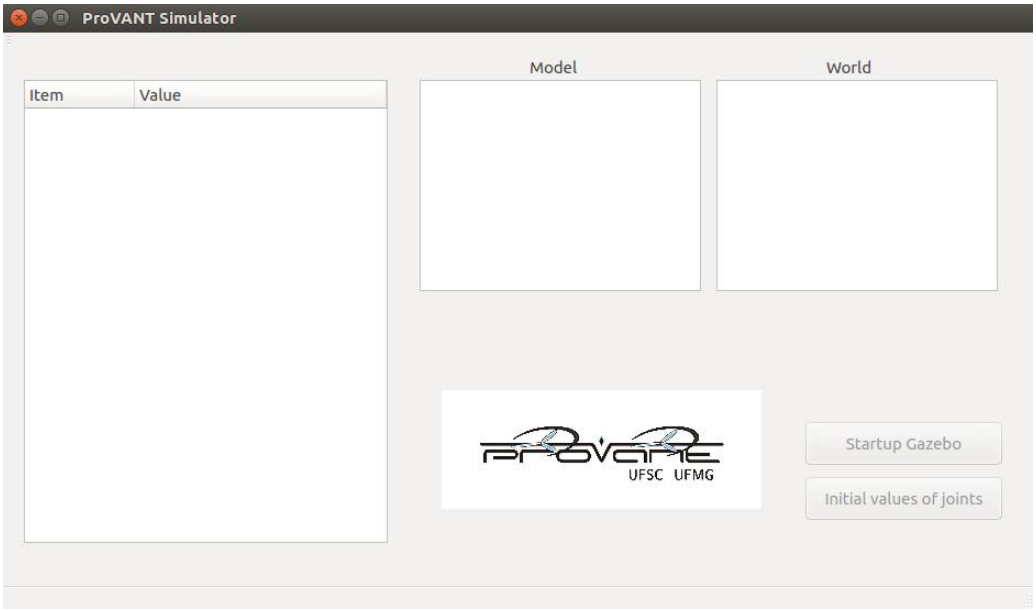

Figura 3.1: Janela inicial.

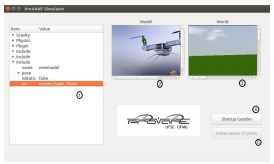

Figura 3.2: Menu superior.

3.3 Seleção do VANT

Através da opção "Edit", no menu superior, é possível selecionar modelo de VANT a ser utilizado na simulação. Atenção: nessa versão do ambiente de simulação, é possível apenas a adição de de um único VANT na mesma instância de simulação.

3.4 Janela de configuração da simulação

Tendo escolhido o VANT e o cenário de simulação, suas imagens a imagem ilustrativa do cenário aparecerá na interface e portanto é possível verificar se foram selecionados corretamente, como mostra a Figura 3.3, através dos itens (2) e (3), respectivamente.

Ainda com relação à Figura 3.3, o item (1) corresponde à uma árvore de informações e configurações do VANT e do cenário de simulação. Através de duplo clique em cima do campo desejado é possível realizar configurações. Os campos dessa arvore são:

- Gravity: vetor de aceleração da gravidade com relação ao referencial do mundo em m/s^2;
- Physics: motor físico física utilizada para realizar simulação. As opções possíveis são:
  - ode (propositório de criação: dinâmica simplificada para robôs)
  - bullet (propositório de criação: jogos)
  - dart: (propositório de criação: computação gráfica e controle de robôs)
  - simbody (propositório de criação: plicações biomecânicas)
- name: Nome do modelo do VANT no simulador Gazebo.;
- pose: Posição (x,y,z) e orientação (roll,pitch,yaw) iniciais do VANT, em m e rad, respectivamente.

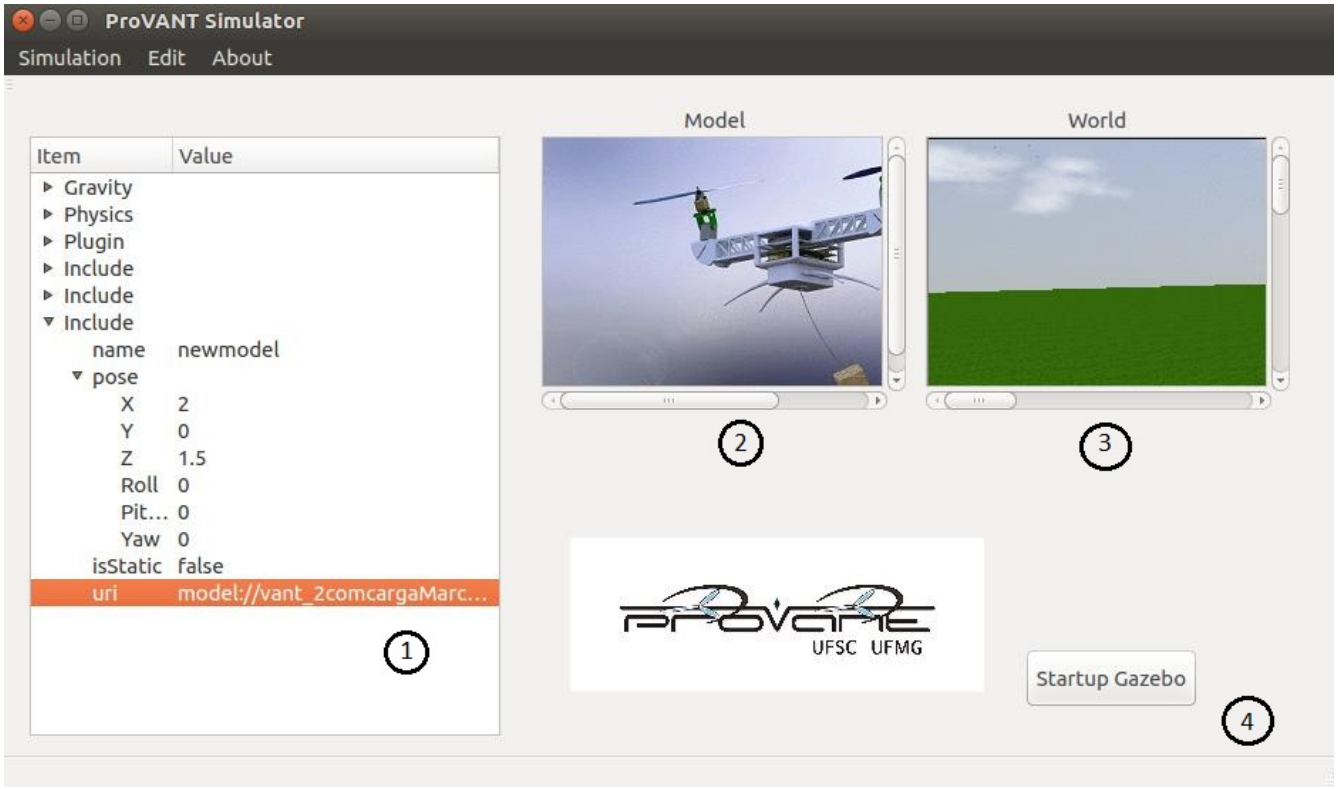

Figura 3.3: Janela de configuração da simulação.

Através do duplo clique no campo uri, uma nova janela se abrirá, na qual é possível visualizar e realizar configurações no modelo, controlador, atuadores e sensores. Mais detalhes sobre está janela são apresentados na próxima seção. O item (4) permite a inicialização da simulação com as configurações, modelo e cenário selecionados pelo usuário na interface gráfica. O item (5) permite a definição de condições iniciais da junta um fez que o mundo e o modelo estejam pronto para simulação.

3.5 Janela de visualização e configuração dos parâmetros do modelo, estratégia de controle e instrumentação

Como mencionado, através de duplo clique no campo uri uma nova janela de configurações será aberta, como mostra a Figura 3.4. Através das quatro abas disponíveis: Parameters, Controller, Sensors e Actuators; é possível gerenciar as configurações do modelo, estratégia de controle e instrumentação utilizados para simulação. A janela é inicializada com a aba Parameters selecionada. Esta aba permite a visualização dos parâmetros do modelo do VANT utilizado na simulação.

A Figura 3.5 mostra a aba "Controller". Nela é possível criar uma nova estratégia de controle, utilizando o item (2), selecionar uma existente, através do item (3) ou compilá-las, item (4). A estratégia de controle selecionada para ser utilizada na simulação é mostrada no item (1). Mais detalhes sobre esses itens são descritos a seguir.

Criando uma nova estratégia de controle

Para criar uma nova estratégia de controle, deve-se clicar no botão "New controller" item (2), e uma nova janela será mostrada (Figura 3.6). Nesta janela, o usuário deve inserir o nome da nova estratégia de controle (mais detalhes sobre a criação de uma nova estratégia de controle são apresentados no Capítulo 4).

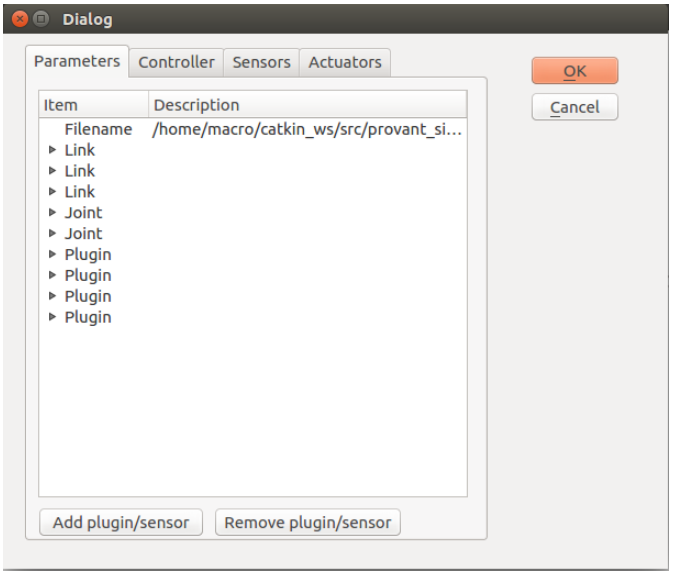

Figura 3.4: Aba para visualização dos parâmetros do modelo do VANT.

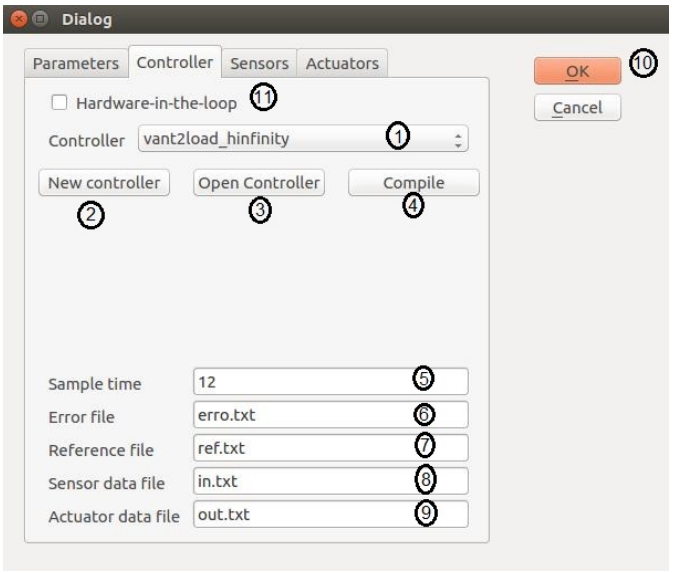

Figura 3.5: Aba para seleção e configuração da estratégia de controle.

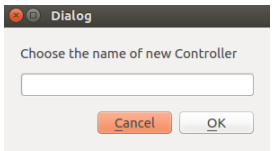

Figura 3.6: Criando uma nova estratégia de controle.

Modificando uma estratégia de controle já existente

Para modificar uma estratégia de controle já existente, o usuário deve selecionar o nome da estrategia de controle entre várias listadas na caixa de listagem item (1) e clicar na botão "Open controller"item (3). Após escolher a estratégia, o gerenciador de arquivos Nautilus será aberto no diretório com todos os arquivos e diretórios associados ao controlador, conforme mostra a Figura 3.7.

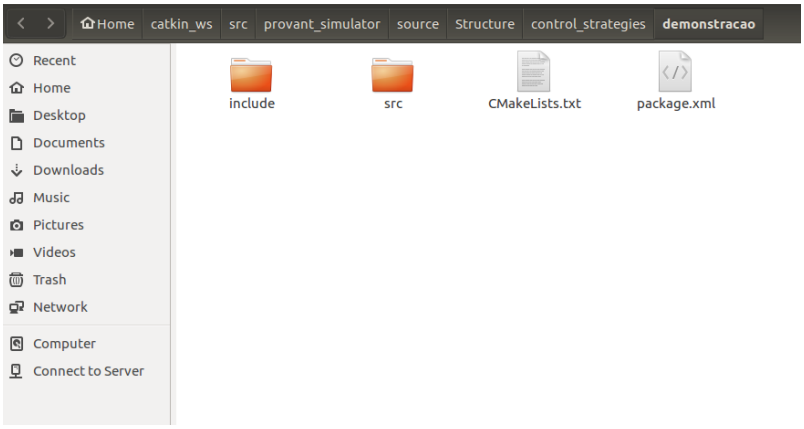

Figura 3.7: Diretório contendo arquivos e pastas associados à estratégia de controle a ser modificada.

Compilando o controlador

Para compilar o código associado à estratégia de controle selecionada, o usuário deve clicar no botão "Compile" item (4). **Esta etapa deve ser realizada sempre que haja modificações na estratégia de controle**. Caso ocorram erros durante a compilação, o relatório de saída do compilador será mostrado em um arquivo de texto por meio do aplicativo gedit.

Alterando configurações adicionais

A aba ”Controller” também permite a configuração de outros parâmetros associados à simulação. O campo ”Sample time”, item (5), permite a configuração do perído de amostragem em milissegundos. Já os demais campos, ”Error file”, item (6), ”Reference data file”, item (7), ”Sensor file”, item (8) e ”Actuator data file”, item (9) determinam o nome dos arquivos de texto onde serão registrados os valores do erro dos estados, trajetória desejada, dados dos sensores e sinais de controle, respectivamente. Estes arquivos podem ser carregados diretamente no MATLAB. Tais arquivos estarão disponíveis no diretório:

```
$HOME/catkin_ws/srcProVANT-Simulator/source/Structure/Matlab
```

3.5.1 Definindo que a simulação será via Hardware-in-the-loop

A partir da caixa de seleção do item (11) o usuário informa que a simulação rodará via Hardware-the-loop.

3.5.2 Selecionando a instrumentação disponível

As abas Sensors e Actuators, ilustradas na Figura 3.8, listam, respectivamente, os nomes dos tópicos dos sensores e atuadores que o controlador terá acesso durante a simulação **de acordo a ordem apresentada**. Tais tópicos são configurados durante a configuração dos plugins, sendo melhor detalhados na seção A.1.2.

Conforme necessário, o usuário deve adicionar, remover e editar os intrumentos (atuadores/sensores) disponíveis na lista. Para adicionar novos instrumentos basta selecionar o botão ”Add” e clicar duas vezes no nome do instrumento desejado. Para removê-los, basta selecionar o instrumento e pressionar o botão ”Remove”.

3.5.3 Inicializando a simulação

Após realizar todas as configurações, o usuário deve pressionar o botão "OK" item (10). A janela de configuração da simulação (Figura 3.3) será mostrada novamente. A simulação pode ser então inicializada com o modelo de VANT, cenário, estratégia de controle e instrumentação selecionados, através do botão "Startup Gazebo" , item (4) da Figura 3.3. O simulador Gazebo será inicializado, conforme mostrado na Figura 3.9.

Para dar início à simulação, o usuário deve pressionar o botão STEP. **O botão PLAY não deve ser utilizado**.

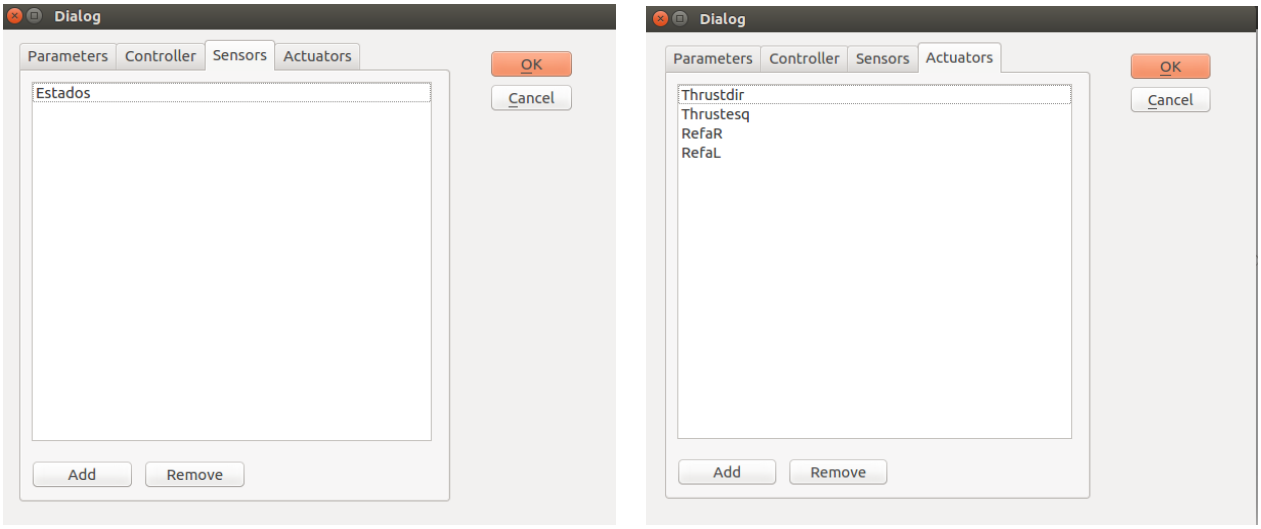

Figura 3.8: Abas para seleção da instrumentação disponível durante a simulação.

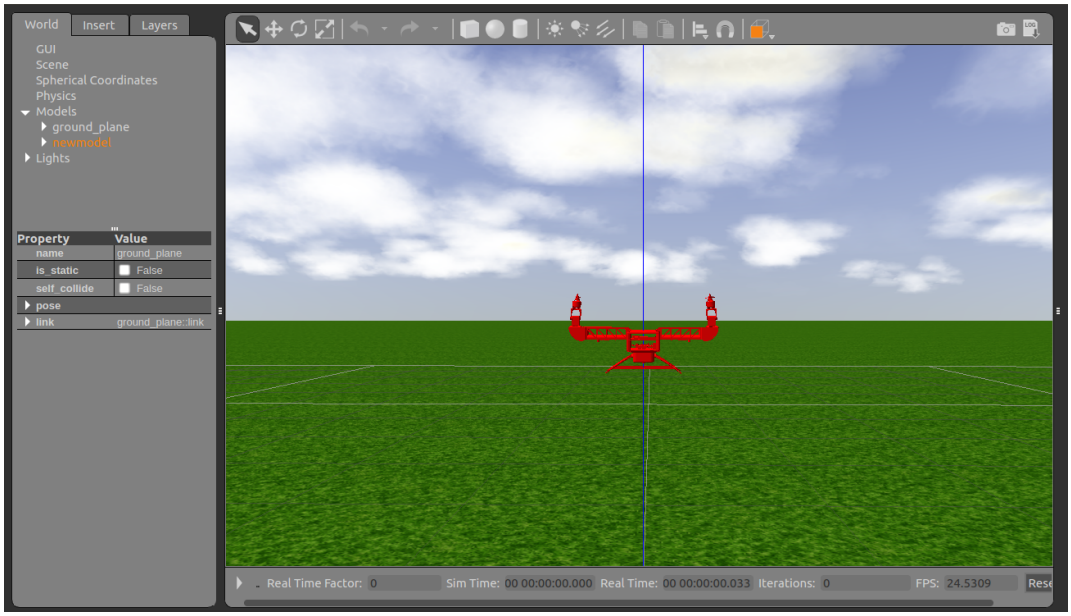

Figura 3.9: Janela inicial do simulador Gazebo.

Capítulo 4

Projeto de estratégia de controle

Neste capítulo são detalhados os procedimentos necessários para implementação de estratégias de controle no ambiente de simulação. Inicialmente é descrita a organização e a estrutura dos arquivos necessários para implementação da estratégia de controle. Por fim, apresenta-se o processo de criação de uma nova estratégia de controle com a finalidade de ilustrar o procedimento apresentado na Seção 3.5.

4.1 Organização

De modo geral a estrutura de arquivos do projeto de controle está organizada como illustrado na Figura 4.1:

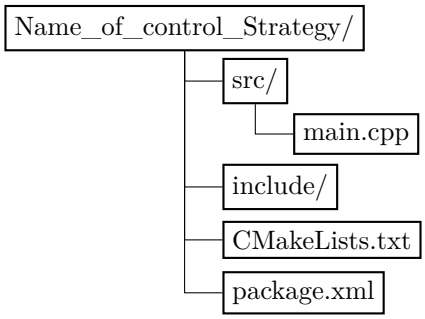

Figura 4.1: Organização do diretório de projeto de estratégias de controle. Os diretórios são dados pelas ”caixinhas” com nomes terminados pelo caractere ”/” e os arquivos possuem alguma extensão em seu nome.

- O arquivo ”main.cpp” é o arquivo onde é implementada a lógica da estratégia de controle.
- A pasta ”include” armazena bibliotecas customizadas pelo usuário que são incluídas no preâmbulo do arquivo main.cpp.
- O arquivo ”CMakeLists.txt” fornece ao compilador informações sobre o diretório de bibliotecas incluídas nos códigos fontes do simulador. Detalhes sobre como incluir novas bibliotecas podem ser encontrados no Apêndice C.
- O arquivo ”package.xml” é o arquivo necessário para armazenar dados como nome do autor, endereço de e-mail, etc. Detalhes de sua configuração são apresentados no Apêndice D.

4.2 Interface padrão para desenvolvimento de estratégias de controle e template main.cpp

A criação de uma estratégia de controle é realizada através de herança da classe virtual padrão "IController". Sendo essa uma classe virtual, seus métodos precisam ser implementadas na classe filha. A Figura 4.2 ilustra suas funções métodos.

A função método *config()* é executada no inicio das simulações, sendo utilizada para realizar configurações iniciais da estratégia de controle. O método *execute()* é chamado pelo simulador a cada período de amostragem. Ele é **o método que deve conter toda a lógica da estratégia de controle**. A ordem e quantidade dos sinais entrada e saída desta função é determinada na interface como descrito na subseção 3.5.2. As funções *state()*, *error()* e *reference()* são métodos que retornam os valores dos sinais de erro, referência e sensores para serem armazenados em arquivos de texto. Os dados salvos nos arquivos txt podem ser utilizado para produzir gráficos dos resultados da simulação, utilizando o matlab, por exemplo.

```
#ifndef ICONTROLLER_HPP
#define ICONTROLLER_HPP

#include "simulator_msgs/SensorArray.h"

class Icontroller
{
        public:
        Icontroller(){};
        virtual ~Icontroller(){};
        virtual void config()=0;
        virtual std::vector<double> execute(simulator_msgs::SensorArray)=0;
        virtual std::vector<double> Reference()=0;
        virtual std::vector<double> Error()=0;
        virtual std::vector<double> State()=0;
};

extern "C" {
        typedef Icontroller* create_t();
        typedef void destroy_t(Icontroller*);
}
#endif
```

Figura 4.2: Interface de implementação de estratégias de controle.

Quando uma nova estratégia de controle é criada, o ambiente de simulação fornece um arquivo main.cpp como template básico para a implementação da estratégia de controle. O conteúdo deste arquivo está ilustrado na Figura 4.3. A próxima seção apresenta um exemplo de implementação de estratégia de controle.

4.3 Exemplo de implementação de estratégia de controle

Esta seção demonstra o processo de implementação de uma nova estratégia de controle. Nela é apresentado um exemplo de utilização da interface gráfica para a criação e modificação de uma nova estratégia de controle, e também o processo de compilação. O exemplo utilizado corresponde à implementação de uma estratégia de controle robusto para rastreamento de trajetória da carga, transportada por um VANT na configuração Tilt-rotor. Mais detalhes da estratégia de controle pode ser encontrados em Lara et al. (2017).

4.3.1 Configurando a lista de sensores e atuadores disponíveis

Na janela descrita na Seção 3.5, ao selecionar a aba Sensors ou Actuator, uma das janelas illustrada na Figura 4.4 aparecerá. Em ambas, o usuário pode adicionar e remover instrumentos da lista utilizando os botões "Add" e "Remove". Além disso, clicando duas vezes sobre um nome na lista, é possível editar as propriedades do instrumento. Estas listas são de fundamental importância para a implementação da estratégia de controle, tais instrumentos e sua respectiva ordem definirão os dados de entrada e saída do controlador.

Neste exemplo, o controlador receberá um vetor de sensores, no entanto contendo informações fornecidas por um único sensor, cujo tópico de comunicação é denominado "Estados". Além disso, o controlador deverá retornar um vetor de ponto flutuante de dimensão 4 com sinais de entrada de controle para atuadores, cujos tópicos de comunicação estão na seguinte ordem: 1) Thrustdir, 2) Thrustesq, 3) RefaR e 4) RefaL.

```
#include "Icontroller.hpp"

class demonstracao : public Icontroller
{
        public: demonstracao(){}
        public: ~demonstracao(){}
        public: void config(){}
        public: std::vector<double> execute(simulator_msgs::SensorArray arraymsg)
        {
                std::vector<double> out;
                return out;
        }
        public: std::vector<double> Reference()
        {
                std::vector<double> out;
                return out;
        }
        public: std::vector<double> Error()
        {
                std::vector<double> out;
                return out;
        }
        public: std::vector<double> State()
        {
                std::vector<double> out;
                return out;
        }
};

extern "C"
{
        Icontroller *create(void) {return new demonstracao;}
        void destroy(Icontroller *p) {delete p;}
}
```

Figura 4.3: Template para implementação de estratégias de controle.

4.3.2 Criando uma nova estratégia de controle

Para criar uma nova estratégia de controle, na aba Controller, pressione ”New Controller”. Uma nova janela será mostrada solicitando ao usuário o nome do novo controlador. Neste exemplo, a estratégia de controle terá o nome ”vant2load_hinfinity” (Figura 4.5). Depois de confirmar o nome do controlador, uma janela do Nautilus aparecerá com o diretório do projeto criado (Figura 4.6).

4.3.3 Implementação da estratégia de controle

A Figura 4.7 mostra um exemplo de código de estratégia de controle implementada no arquivo main.cpp. No exemplo, observa-se que é necessário a inclusão de 3 bibliotecas:

- #include ”Icontroller.hpp” - informa ao código a interface padrão para criação de controladores no ambiente de simulação;
- #include <Eigen/Eigen> - importa as funcionalidades fornecidas pela biblioteca Eigen [1]. A biblioteca Eigen fornecer funções para realizar operações da álgebra linear;
- #include ”simulator_msgs/Sensor.h” - informa a classe responsável por abstrair o padrão de comunicação entre o controlador e os sensores.

[1] https://eigen.tuxfamily.org

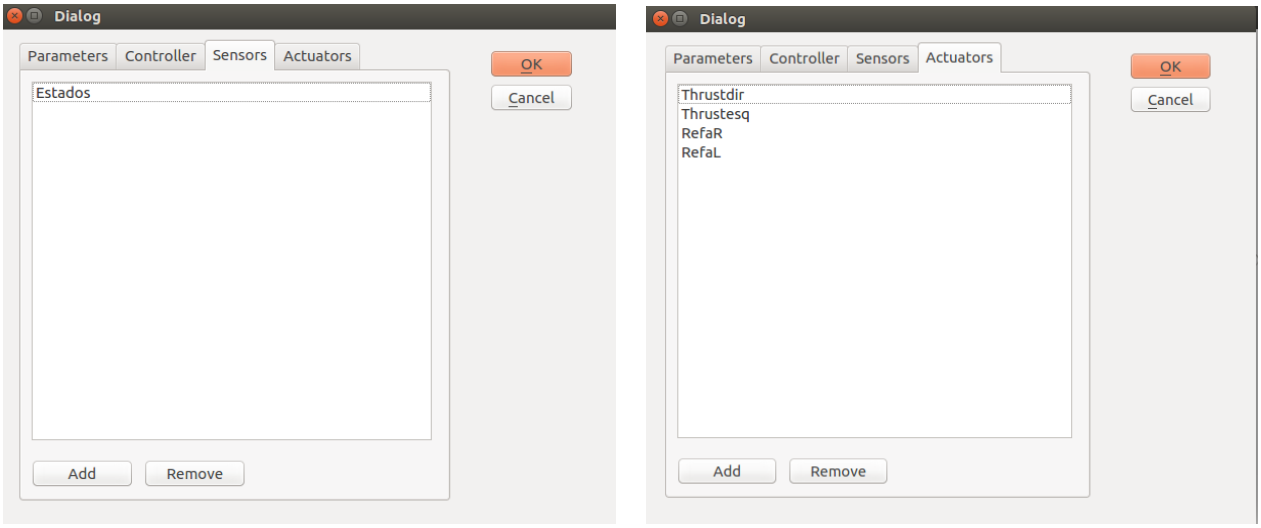

Figura 4.4: Abas Sensors e Actuators.

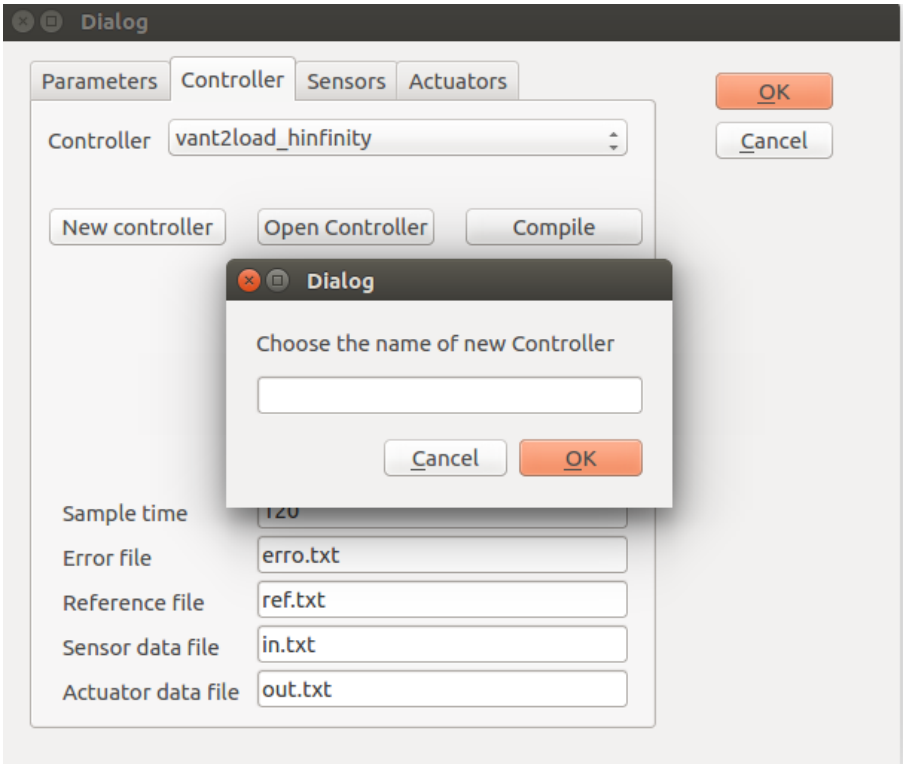

Figura 4.5: Criando uma nova estratégia de controle.

Os atributos **Xref**, **Erro**, **Input**, **K** e **X** são as estruturas de de vetores e matrizes da biblioteca Eigen. Essas estruturas são responsáveis por realizar a operações de álgebra linear e, portanto, permitir a relização da lei de controle. O atributo **T** determina o período de amostragem do controlador.

O construtor hinfinity() inicializa os atributos da classe, enquanto o destrutor ~hinfinity() não possui nenhuma função no contexto do simulador.

Neste exemplo, o método config() é utilizado para atribuir valores à matriz de ganhos K, respeitando a sintaxe da biblioteca Eigen, no entanto o usuário pode utilizar esse espaço para realizar qualquer outra configuração inicial. Por fim, a lógica da estratégia de controle é implementada no método execute(), que é executado a cada período de amostragem, como descrito anteriormente.

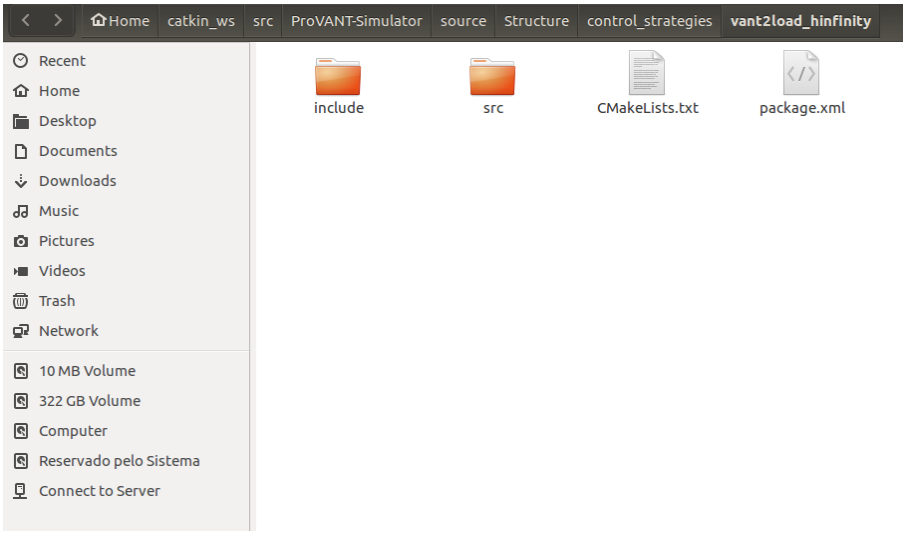

Figura 4.6: Arquivos e diretórios associados à nova estratégia de controle.

O código começa por declarar as variáveis (estáticas) ”xint”, ”x_ant”, ”yint”, ”y_ant”, ”zint”, ”z_ant”, ”yawint” e ”yaw_ant”, utilizadas para armazenar os valores dos integradores implementados na estratégia de controle. Em seguida, está presente o trecho de código referente aos dados dos sensores, obtidos através do vetor ”arraymsg”. Como neste exemplo somente um sensor está disponível (”Estados”), apenas a primeira posição deste vetor é acessada (através da sintaxe ”arraymsg.values.at(0)”). Nos casos em que mais sensores são disponibilizados, os dados do n-ésimo sensor são acessados através de ”arraymsg.values.at(n-1)” e a ordem desse vetor é definida a priori na aba Sensor conforme é visualizada em 3.8. Em seguida, é definido a referência para o controlador e implementado integradores, através do método de integração trapezoidal, para realização da lei de controle e, por fim, é calculada a ação de controle feedforward.

Os métodos Reference(), Error(), State() são utilizados para armazenar dos sinais de referência, erro e estados, respectivamente, nos arquivos de texto de saída do ambiente de simulação. A função pqr2EtaDot() corresponde a um mapeamento de parte das informações recebidas pelos sensores, necessário para a implementação da estratégia de controle deste exemplo.

Assim como a função pqr2EtaDot(), quaisquer funções adicionais que sejam necessárias para a implementação da estratégia de controle, podendo estar relacionadas também a algoritmos de filtragem, podem ser escritas dentro da classe ”main.cpp”, ou em classes auxiliares criadas dentro da pasta ”include” ilustrada em 4.6 (neste caso, estas devem ser também incluídas no cabeçalho do arquivo ”main.cpp”).

Por fim, o trecho de código a seguir corresponde à funções necessárias para garantir o carregamento da estratégia de controle em tempo de execução do simulador. O método create() cria uma instância da classe que encapsula o controlador, e o método destroy() é utilizado para destruir a instância da classe.

```
extern "C"
{
        Icontroller *create(void) {
                return new hinfinity;
        }
        void destroy(Icontroller *p) {
                delete p;
        }
}
```

4.3.4 Compilando o código da estratégia de controle

Após a implementação da estratégia de controle, o código correspondente pode ser compilado através do botão ”Compile” (conforme explicado no Capítulo 2, vide Figura 3.5, item 4).

```
#include "Icontroller.hpp"
#include <Eigen/Eigen>
#include "simulator_msgs/Sensor.h"

class hinfinity : public Icontroller
{
        private: Eigen::VectorXd Xref; // vetor de referência
        private: Eigen::VectorXd Erro; // vetor de erros
        private: Eigen::VectorXd Input; // sinais de controle
        private: Eigen::MatrixXd K; // matriz de ganhos do controlador
        private: Eigen::VectorXd X; // vetor de estados
        private: double T; // Período de amostragem

        public: hinfinity(): Xref(24), K(4,24), X(24), Erro(24), Input(4)
        {
                T = 0.012;
        }
        public: ~hinfinity(){        }

        public: void config()
        {
                K<< [...] Dados da matriz [...];
        }
        public: std::vector<double> execute(simulator_msgs::SensorArray arraymsg)
        {
                static float count = 0;
                static float xint, x_ant = 0;
                static float yint, y_ant = 0;
                static float zint, z_ant = 0;
                static float yawint, yaw_ant = 0;
                // selecionando dados
                int i = 0;
                simulator_msgs::Sensor msgstates;
                msgstates = arraymsg.values.at(0);
                // Referência
                float trajectoryRadius = 2;
                float trajectoryHeight = 4*trajectoryRadius;
                float trajTime = 80;
                float pi = 3.14;
                float x = trajectoryRadius*cos((count*T)*2*pi/trajTime);
                float xdot = -trajectoryRadius*(2*pi/trajTime)*sin((count*T)*2*pi/trajTime);
                float xddot = -trajectoryRadius*(2*pi/trajTime)*(2*pi/trajTime)
                                *cos((count*T)*2*pi/trajTime);
                float y = trajectoryRadius*sin((count*T)*2*pi/trajTime);
                float ydot = trajectoryRadius*(2*pi/trajTime)*cos((count*T)*2*pi/trajTime);
                float yddot = -trajectoryRadius*(2*pi/trajTime)*(2*pi/trajTime)
                                *sin((count*T)*2*pi/trajTime);
                float z = trajectoryHeight+1 - trajectoryHeight*cos((count*T)*2*pi/trajTime);
                float zdot = trajectoryHeight*(2*pi/trajTime)*sin((count*T)*2*pi/trajTime);
                float zddot = trajectoryHeight*(2*pi/trajTime)*(2*pi/trajTime)
                                *cos((count*T)*2*pi/trajTime);
                Xref << x,y,z,0,0,0,0.00002965,0.004885,0.004893,0.00484,xdot,ydot,zdot,
                        0,0,0,0,0,0,0,0,0,0,0;
```

```
//Convertendo velocidade angular
std::vector<double> etadot = pqr2EtaDot(msgstates.values.at(13),
msgstates.values.at(14),
msgstates.values.at(15),
msgstates.values.at(3),
msgstates.values.at(4),
msgstates.values.at(5));

// Integrador Trapezoidal
float x_atual = msgstates.values.at(0) - Xref(0);
xint = xint + (T/2)*(x_atual + x_ant);
x_ant = x_atual;
float y_atual = msgstates.values.at(1) - Xref(1);
yint = yint + (T/2)*(y_atual + y_ant);
y_ant = y_atual;
float z_atual = msgstates.values.at(2) - Xref(2);
zint = zint + (T/2)*(z_atual + z_ant);
z_ant = z_atual;
float yaw_atual = msgstates.values.at(5) - Xref(5);
yawint = yawint + (T/2)*(yaw_atual + yaw_ant);
yaw_ant = yaw_atual;

// vetor de estados aumentado
X << msgstates.values.at(0),//x
msgstates.values.at(1),//y
msgstates.values.at(2),//z
msgstates.values.at(3),//roll
msgstates.values.at(4),//pitch
msgstates.values.at(5),//yaw
msgstates.values.at(8),//g1 x
msgstates.values.at(9),//g2 y
msgstates.values.at(6),//aR
msgstates.values.at(7),//aL
msgstates.values.at(10),//vx
msgstates.values.at(11),//vy
msgstates.values.at(12),//vz
etadot.at(0),//droll
etadot.at(1),//dpitch
etadot.at(2),//dyaw
msgstates.values.at(18), // angulo da primeira junta da carga
msgstates.values.at(19), // angulo da segunda junta da carga
msgstates.values.at(16), // derivada do angulo da primeira junta da carga
msgstates.values.at(17), // derivada do angulo da segunda junta da carga
xint,
yint,
zint,
yawint;


// lei de controle
Erro = X-Xref;
Input = -K*Erro;
```

```
                // Criando vetor com dados da trajetória de referência
                Eigen::MatrixXd qref(10,1);
                qref << x,y,z,0,0,0,0.00002965,0.004885,0.004893,0.00484;
                Eigen::MatrixXd qrefdot(10,1);
                qrefdot << xdot,ydot,zdot,0,0,0,0,0,0,0;
                Eigen::MatrixXd qrefddot(10,1);
                qrefddot << xddot,yddot,zddot,0,0,0,0,0,0,0;

                // Feedforward
                Input(0) = Input(0) + 12.6005;
                Input(1) = Input(1) + 12.609;
                count++;


                // convertendo dados do vetor da biblioteca Eigen para vetor a biblioteca padrão de C++
                std::vector<float> out2(Input.data(), Input.data() + Input.rows()
                                        * Input.cols());

                std::vector<double> out(out2.size());
                for(int i=0; i<out2.size();i++) out.at(i) = out2.at(i);
                return out;
        }

        public: std::vector<double> Reference()
        {
                std::vector<double> out(Xref.data(), Xref.data() + Xref.rows()
                                        * Xref.cols());
                return out;
        }

        public: std::vector<double> Error()
        {
                std::vector<double> out(Erro.data(), Erro.data() + Erro.rows()
                                        * Erro.cols());
                return out;
        }

        public: std::vector<double> State()
        {
                std::vector<double> out(X.data(), X.data() + X.rows() * X.cols());
                return out;
        }

        private: std::vector<double> pqr2EtaDot(double in_a, double in_b, double in_c,
                                                double phi, double theta, double psii)
        {
                std::vector<double> out;
                out.push_back(in_a + in_c*cos(phi)*tan(theta) + in_b*sin(phi)*tan(theta));
                out.push_back(in_b*cos(phi) - in_c*sin(phi));
                out.push_back((in_c*cos(phi))/cos(theta) + (in_b*sin(phi))/cos(theta));
                return out;
        }
};
```

```
extern "C"
{
        Icontroller *create(void) {
                return new hinfinity;
        }
        void destroy(Icontroller *p) {
                delete p;
        }
}
```

Figura 4.7: Exemplo de código.

Apêndice A

Modelos

Os arquivos associados aos modelos de VANTs, utilizados no ambiente de simulação ProVANT, estão localizados na pasta referênte ao caminho:

```
$HOME/catkin\_ws/src/provant\_simulator/source/Database/simulation_elements/models/real
```

Cada modelo no ambiente de simulação ProVant possui um diretório com o seu respectivo nome. Nesse diretório estão arquivos que descrevem os modelos dinâmicos, visuais, de colisão, sensoriais e da lei de controle utilizada. Portanto, caso seja necessário adicionar um novo modelo, ou editar um existente, o mesmo deve possuir(ou possui) arquivos de configuração/descrição do VANT organizados conforme os diretórios ilustrados na Figura A.1., onde

1. O arquivo config.xml armazena as informações referentes ao controlador a ser utilizado no modelo.
2. O arquivo model.sdf descreve o modelo dinâmico, de colisão e visual do VANT para o simulador Gazebo.
3. O arquivo model.config descreve metadados do modelo.
4. O arquivo imagem.gif contém a imagem utilizada pela interface gráfica para ilustrar o modelo do VANT (vide Figura 3.3, item 2).
5. O diretório meshes armazena os arquivos exportados da ferramenta CAD (utilizada para o desenvolvimento mecânico do VANT), exemplo: SolidWorks.

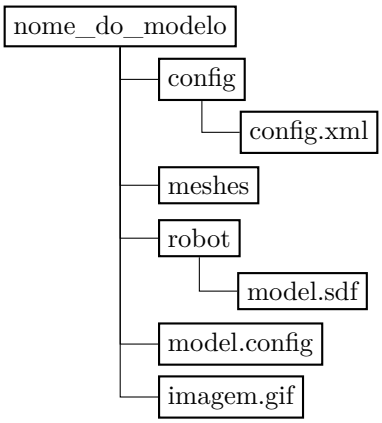

Figura A.1: Organização do diretório com arquivos de descrição do modelo VANT

No arquivo ”model.config” o gazebo identifica onde está o arquivo com os dados estruturais do modelo, além de informações associadas à autoria, versão e descrição do modelo. O Código A.1 ilustra um exemplo desse arquivo.

```
<?xml version="1.0"?>
<model>
        <name>vant</name>
        <version>1.0</version>
        <sdf version='1.5'>robot/model.sdf</sdf>
        <author>
                <name>provant</name>
                <email>provant@ufmg.br</email>
        </author>
        <description>
                The UAV version 3.0 of the provant project
        </description>
</model>
```

Código A.1: Exemplo de conteúdo existente no arquivo "model.config".

As *tags* utilizadas no arquivo "model.config" são:

- <name></name>: especifica o nome do modelo;
- <version></version>: especifica a versão;
- <sdf></sdf>: especifica o arquivo com a descrição do modelo dinâmico, de colisão e visual de modelo para o simulador Gazebo;
- <author><name></name></author>: especifica o nome do autor do modelo;
- <author><email></email></author>: especifica o email para contato com o autor;
- <description></description>: descreve brevemente o modelo.

O arquivo "config.xml" armazena todas as configurações relativas à estrutura de controle que será utilizada durante a simulação, como: sensores, atuadores, lei de controle e período de amostragem. Com o intuíto de ajudar o usuário com a edição desse arquivo, a interface gráfica fornece ferramentas para esse propósito, não sendo necessário que o usuário acesse diretamente o arquivo. O Código A.2: ilustra um exemplo desse arquivo.

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
        <TopicoStep>Step</TopicoStep>
        <Sampletime>12</Sampletime>
        <Strategy>libvant3_lqr.so</Strategy>
        <RefPath>ref.txt</RefPath>
        <Outputfile>out.txt</Outputfile>
        <InputPath>in.txt</InputPath>
        <ErroPath>erro.txt</ErroPath>
        <Sensors>
                <Device>Estados</Device>
        </Sensors>
        <Actuators>
                <Device>ThrustR</Device>
                <Device>ThrustL</Device>
                <Device>TauR</Device>
                <Device>TauL</Device>
                <Device>Elevdef</Device>
                <Device>Ruddef</Device>
        </Actuators>
</config>
```

Código A.2: Exemplo de arquivo "config.xml".

As *tags* utilizadas no arquivo "config.xml" são:

- <TopicoStep></TopicoStep>: declara tópico de sincronização do simulador ao controlador. **O valor desta tag não deve ser alterado**.;
- <Sampletime></Sampletime>: define o período de amostragem do controlador, em milissegundos;
- <Strategy></Strategy>: nome da biblioteca dinâmica associada à implementação da estratégia de controle a ser utilizada em conjunto com esse modelo;
- <RefPath></RefPath>: nome do arquivo no qual os dados do sinal de referência são armazenados;
- <Outputfile></Outputfile> nome do arquivo no qual os dados dos sinais de controle são armazenados;
- <InputPath></InputPath>: nome do arquivo no qual os dados dos sensores são armazenados;
- <ErroPath></ErroPath> nome do arquivo no qual os dados do vetor de erro são armazenados;
- <Sensors></Sensors> nome dos tópicos dos sensores aos quais a estratégia de controle terá acesso (na ordem especificada)
- <Actuators></Actuators> nome dos tópicos dos atuadores aos quais a estratégia de controle terá acesso. (o controlador deve retornar um vetor com a mesma quantidade de dados e na ordem aqui especificados)

O arquivo "model.sdf" fornece ao Gazebo as informações do modelo do VANT no formato XML conforme o padrão SDF [1] (A Figura **??** illustra um exemplo desse tipo de arquivo). A próxima seção descreve com mais detalhes a estrutura básica de um arquivo SDF.

A.1 O arquivo SDF

Antes de apresentar a configuração básica de um arquivo SDF, é necessário primeiro introduzir algums conceitos: No simulador um modelo corresponde a um sistema mecânico, que pode ser formado por um ou múltiplos corpos rígidos[2]. Assim, como em um manipulador, no simulador os corpos são denominados elos, sendo os elos filhos conectados a elos pai através de juntas. Elos filhos são corpos rígidos que possuem movimento restringido pela conexão ("junta") com corpos denominados elos pai. Os elos possuem propriedades inerciais, visuais e de colisão. Já as juntas, impõem a restrição do movimento relativo entre dois elos, com propriedades como o tipo de junta (Prismática, rotativa, etc.), limites de movimento (Posição e velocidade), existência de atrito, etc. O Código A.3 ilustra um exemplo de arquivo SDF.

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf version="1.4">
        <model name="modelo">
                <link name="corpo">
                        ...
                </link>
                <link name="servo">
                        ...
                </link>
                <joint name="corpo_servo">
                        ...
                </joint>
        </model>
</sdf>
```

Código A.3: Descrição de um elo no arquivo "modelo.sdf"

As *tags* utilizadas são:

- <link></link>: especifica a existência de um elo, com o seu nome;
- <joint></joint>: especifica a existência de uma junta, com o seu nome.

Cada elo do modelo possui três tipos de descrições para o simulador: cinemática, visual e de colisão. A estrutura de configuração de um elo em um arquivo SDF possui o formato ilustrado no Código A.4, onde:

- <pose></pose>: define a pose do elo;
- <inertial></inertial>: especifica as propriedades inerciais do elo;

[1] http://sdformat.org/spec
[2] Ao assumir um corpo como rígido são desprezando efeitos de elasticidade e deformações

- <collision></collision>: especifica o modelo de colisão do elo. Os modelos de colisão dos VANTs usados no ambiente de simulação são obtidos de arquivos CAD;
- <visual></visual>: especifica características visuais, como cor e formato. Os modelos visuais dos VANTs usados no ambiente de simulação são obtidos de arquivos CAD, exceto a cor, que é especificada separadamente.

```
<link name="servodir">
        <pose>0.02E-3 -277.61E-3 56.21E-3 -0.0872665 0 0</pose>
        <inertial>
                ...
        </inertial>
        <collision name="servodircollision"> <!--opcional-->
                ...
        </collision>
        <visual name="servodirvisual"> <!--opcional-->
                ...
        </visual>
</link>
```

Código A.4: Descrição da de um elo no "modelo.sdf"

Parâmetros de inércia do elo: O usuário deve informar os parâmetros de inércia de cada elo na *tag* "inertial". As informações obrigatórias são a massa, posição relativa do centro de massa e o tensor de inércia. No Código A.5 é ilustrado um exemplo de configuração dos parâmetros de inércia de um elo em formato SDF, onde:

- <mass></mass>: define a massa do elo;
- <pose></pose>: especifica a posição do centro de massa do elo em relação a seu sistema de coordenadas principal;
- <inertia></inertia>: especifica o tensor de inércia do elo;

```
<inertial>
        <mass>0.0809439719362664</mass>
        <pose>
                -3.60859273452335E-10 -0.000226380714807978 0.0594780519701684 0 0 0
        </pose>
        <inertia>
                <ixx>3.88267747087835E-06</ixx>
                <ixy>6.03219085082653E-06</ixy>
                <ixz>-2.78471406661236E-12</ixz>
                <iyy>0.000104858690365283</iyy>
                <iyz>7.0486590219062E-07</iyz>
                <izz>8.31755564684115E-05</izz>
        </inertia>
</inertial>
```

Código A.5: Descrição de características inerciais no arquivo "modelo.sdf"

Propriedades de colisão do elo: Para que efeitos de colisão sejam aplicados ao elo, o usuário deve descrever o formato do elo no arquivo "model.sdf". Existem diversas formas de descrição, porém este manual apresenta apenas o método utilizado nos modelos de VANTs do ambiente de simulação ProVANT, que consiste na importação de arquivos criados através de ferramentas CAD, como o SolidWorks. No Código A.6 mostra um exemplo de descrição dos parâmetros visuais de um elo a partir de um arquivo STL, onde:

- <pose></pose>: especifica a pose do modelo colisão em relação ao centro de coordenadas do elo;
- <uri></uri>: caminho do arquivo mesh, a partir do diretório do modelo, obtido via exportação no SolidWorks;

```
<collision name="servodircollision">
        <pose>0 0 0 0 0 0</pose>
        <geometry>
```

```
                <mesh>
                        <uri>model://vant_2comcarga/meshes/servodir.STL</uri>
                </mesh>
        </geometry>
</collision>
```

Código A.6: Descrição de características de colisão no arquivo "model.sdf"

Propriedades visuais do elo: Para que o elo seja visualizado durante a simulação, o usuário deve descrever os parâmetros visuais do elo no arquivo "model.sdf". Assim como no caso anterior, existem diversas formas de descrição, porém este manual ilustra apenas o método utilizado nos modelos de VANTs do ambiente de simulação, que consiste na importação de arquivos criados através de ferramentas CAD. O Código A.7 mostra um exemplo de descrição dos parâmetros visuais de um elo a partir de um arquivo STL, onde:

- <pose></pose>: especifica a pose que o modelo visual do elo será definido em relação ao sistema de coordenadas do elo;
- <uri></uri>: caminho do arquivo mesh, a partir do diretório do modelo, obtido via exportação no SolidWorks;
- <ambient></ambient>: definição de cor ambiente;
- <diffuse></diffuse>: definição de cor difusa;
- <specular></specular>: definição de cor especular;
- <emissive></emissive>: definição de cor emissiva.

```
<visual name="servodirvisual">
        <pose>0 0 0 0 0 0</pose>
        <geometry>
                <mesh>
                        <uri>model://vant_2comcarga/meshes/servodir.STL</uri>
                </mesh>
        </geometry>
        <material>
                <ambient>0 0 0 0</ambient>
                <diffuse>1 1 1 1</diffuse>
                <specular>0.1 0.1 0.1 1</specular>
                <emissive>0 0 0 0</emissive>
        </material>
</visual>
```

Código A.7: Descrição de características visuais no arquivo "model.sdf"

A.1.1 Descrição de junta

Há 7 tipos de juntas no simulador:

- **revolute**: junta rotativa;
- **gearbox**: junta rotativa com presença de engrenagens para transmissão de movimento angular entre elos com diferentes relações de torque e velocidade;
- **revolute2**: junta composta por duas juntas rotativas em série;
- **prismatic**: junta prismática;
- **universal**, junta com comportamento de uma esfera articulada;
- **piston**: junta com comportamento da combinação de uma junta rotativa e uma junta prismática.

Um exemplo de estrutura de configuração de uma junta é mostrado no Código A.8, onde:

- <pose></pose>: descreve a pose relativa em que o elo filho está em relação ao elo pai.

- <parent></parent>: nome do elo pai.
- <child></child>: nome do elo filho.
- <axis></axis>: vetor unitário que corresponde ao eixo de rotação da junta. (expressado no sistema de coordenadas do modelo, se estiver utilizando a versão 1.4 do formato SDF; e expressado no sistema de coordenadas do elo filho, se estiver utilizando a versão 1.6 do formato SDF).
- <lower></lower>: limite inferior de posição da junta (em rad, caso seja do tipo rotativa, e metros, caso seja prismática).
- <upper></upper>: limite superior de posição da junta (em rad, caso seja do tipo rotativa, e metros, caso seja prismática).
- <velocity></velocity>: limite de velocidade da junta (em rad/s, caso seja do tipo rotativa, e m/s, caso seja prismática).
- <effort></effort>: limite de esforço da junta (em N.m, caso seja do tipo rotativa, e N, caso seja prismática).
- <damping></damping>: coeficiente de atrito viscoso.
- <friction></friction>: coeficiente de atrito estático.

```
<joint name="aR" type="revolute">
        <pose>0 0 0 0 0 0</pose>
        <parent>corpo</parent>
        <child>servodir</child>
        <axis>
                <xyz>0 0.9962 -0.0872</xyz>
                <limit>
                        <lower>-1.5</lower>
                        <upper>1.5</upper>
                        <effort>2</effort>
                        <velocity>0.5</velocity>
                </limit>
                <dynamics>
                        <damping>0</damping>
                        <friction>0</friction>
                </dynamics>
        </axis>
</joint>
```

Código A.8: Descrição de juntas no arquivo "model.sdf"

A.1.2 Descrição de plugins

Plugins são bibliotecas dinâmicas carregadas durante a inicialização do simulador, a partir das configurações armazenadas no arquivo de descrição de modelo (arquivo SDF). Tais bibliotecas são utilizadas para a implementação de sensores e atuadores no ambiente de simulação.

O ambiente de simulação ProVANT utiliza apenas dois tipos de plugins existentes no simulador Gazebo para controle e monitoramento do modelo do VANT: Modelo e Sensor. Mais detalhes sobre os plugins disponibilizados no simulador Gazebo podem ser encontrados em: http://gazebosim.org/tutorials/?tut=plugins_hello_world

Plugins modelo: Plugins modelo são bibliotecas dinâmicas que controlam e monitoram variáveis de simulação do modelo VANT. Através delas, é possível criar sensores e atuadores customizados. Para inserir um plugin modelo no arquivo "model.sdf", o usuário deve adicionar tags <plugin></plugin>, definindo nome, nome da biblioteca dinâmica e as tags internas necessárias para configuração do plugin. O Código A.9 exemplifica o processo de inserção.

Opções de plugins modelo disponíveis no ambiente de simulação ProVANT estão detalhados no apêndice B.

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf version="1.4">
```

```
        <model name="modelo">
                <link name="corpo">
                        ...
                </link>
                <link name="servo">
                        ...
                </link>
                <joint name="corpo_servo">
                        ...
                </joint>
                <plugin name="plugin_servo">
                        ...
                </plugin>
        </model>
</sdf>
```

Código A.9: Inserção de plugins modelo no arquivo ”model.sdf”

Plugins sensor: Plugins sensor são bibliotecas dinâmicas que simulam sensores, portanto, são utilizados pelo ambiente de simulação ProVANT para mensurar dados de um modelo VANT. Para inserir plugins sensor basta inserir tags <sensor></sensor>, definindo nome e as tags internas necessárias para configuração do plugin. O Código A.10 exemplifica o processo de inserção.

```
<link name="servodir">
        <pose>0.02E-3 -277.61E-3 56.21E-3 -0.0872665 0 0</pose>
        <inertial>
                ...
        </inertial>
        <collision name="servodircollision"><!--opcional-->
                ...
        </collision>
        <visual name="servodirvisual"> <!--opcional-->
                ...
        </visual>
        <sensor name="servosensor"> <!--opcional-->
                ...
        </sensor>
        <sensor name="servosensor2"> <!--opcional-->
                ...
        </sensor>
</link>
```

Código A.10: Inserção de plugins sensor no arquivo ”model.sdf”

Os sensores disponíveis no ambiente de simulação ProVANT são GPS, IMU, sonar e magnetômetro (Detalhes de configurações no arquivo model.sdf podem ser encontrados em http://sdformat.org/spec). No entanto, estes sensores não possuem interface de comunicação com o ROS, pois eles transmitem seus dados via tópicos do Gazebo. Para realizar a transmissão destes dados para tópicos do ROS, é necessário adicionar, juntamente aos plugins sensores, plugins modelo. Tais plugins estão especificados no apêndice B.

Obs.: para funcionamento correto do sensor, o usuário deve ajustar a taxa de amostragem exatamente para o inverso do passo de simulação, por exemplo 1000 HZ.

A.1.3 Padrão de mensagens de comunicação

Como padrão, todos o plugins e sensores disponíveis no ambiente de simulação disponibilizam dados através de uma mesma estrutura de dados. Esta estrutura de dados é abstraída no ROS através de mensagens. As mensagens são estruturas de dados simples, contendo campos tipados. A mensagem padrão de plugins sensores está ilustrada abaixo. onde:

- header: fornece o instante que o dado foi obtido, o frame e o número de sequência durante a comunicação.
- name: fornece o nome do instrumento que forneceu a mensagem.
- values: vetor composto pelos mais variados dados providos pelos sensores.

```
Header header
string name
float64[] values
```

Por outro lado o controlador recebe um outro tipo de mensagem que armazena as mensagens de todos os sensores num dado passo de simulação num mesmo local e na ordem pré-definida pelo usuário, como é descrito na seção 3.5.2. Tal estrutura esta ilustrada a seguir.

```
Header header
string name
Sensor[] values
```

Apêndice B

Plugins existentes no ambiente de simulaçao ProVANT

B.1 Plugins modelo

Esta seção mostra a configuração dos plugins modelos que não tem função de transporte de dados entre Gazebo e ROS. Tais informações estão nas Tabelas B.1, B.2, B.3, B.4, B.5, B.6, B.7 e B.8.

Descrição:	plugin para simulação das forças de empuxo resultado do giro das duas hélices pelos motores brushless de um tilt-rotor
Arquivo:	libgazebo_ros_brushless_plugin.so
Configurações:	<topic_FR> </topic_FR> nome do tópico refente ao valor da força a ser aplicada na hélice direita <topic_FL> </topic_FL> nome do tópico refente ao valor da força a ser aplicada na hélice esquerda <LinkDir> </LinkDir> nome do elo correspondente à hélice direita (a força será aplicada no eixo z desse elo) <LinkEsq> </LinkEsq> nome do elo correspondente à hélice esquerda (a força será aplicada no eixo z desse elo)

Tabela B.1: Configuração do plugin motor brushless.

Descrição:	plugin para simulação servo motor com funções de Torque e posição
Arquivo:	libgazebo_servo_motor_plugin.so
Configurações :	<NameOfJoint> </NameOfJoint> nome da junta a ser controlada pelo servo motor <TopicSubscriber> </TopicSubscriber> nome do tópico com valores de referência para o servo motor <LinkDir> </LinkDir> Nome do tópico com valores de sensoriamento do servo (posição e velocidade) <Modo> </Modo> Modo de funcionamento do servo motor (Opções: ”Torque” ou ”Posicao”)

Tabela B.2: Configuração do plugin servo-motor.

B.2 Plugins modelo para uso junto aos plugins Sensors

Esta seção mostra a configuração dos plugins modelos que tem função de transporte de dados entre Gazebo e ROS. Tais informações estão nas tabelas B.9,B.10,B.11 e B.12.

Descrição:	plugin para sensoriamento do vetor de estados de um VANT Tilt-rotor. (x,y,z, ϕ,θ,ψ,α_R,α_L,$\dot{x}$,$\dot{y}$,$\dot{z}$,$\dot{\phi}$,$\dot{\theta}$,$\dot{\psi}$,$\dot{\alpha_R}$,$\dot{\alpha_L}$)
Arquivo:	libgazebo_AllData_plugin.so
Configurações:	<NameOfTopic> </NameOfTopic> nome do tópico para o usuário obter informações <NameOfJointR> </NameOfJointR> nome a junta do servo motor direito <NameOfJointL> </NameOfJointL> nome a junta do servo motor esquerdo <bodyname> </bodyname> nome do elo correspondente ao corpo principal do servo motor

Tabela B.3: Configuração do plugin State space.

Descrição:	plugin para sensoriamento do vetor de estados de um VANT Tilt-rotor com a função transporte de carga. (x,y,z, ϕ,θ,ψ,α_R,α_L,λ_x,λ_y,$\dot{x}$,$\dot{y}$,$\dot{z}$,$\dot{\phi}$,$\dot{\theta}$,$\dot{\psi}$,$\dot{\alpha_R}$,$\dot{\alpha_L}$,$\dot{\lambda_x}$,$\dot{\lambda_y}$)
Arquivo:	libgazebo_AllData2_plugin.so
Configurações:	<NameOfTopic> </NameOfTopic> nome do tópico para o usuário obter informações <NameOfJointR> </NameOfJointR> nome a junta do servo motor direito <NameOfJointL> </NameOfJointL> nome a junta do servo motor esquerdo <NameOfJoint_X> </NameOfJoint_X> nome a junta correspondente ao grau de liberdade da carga em torno do eixo X <NameOfJoint_Y> </NameOfJoint_Y> nome a junta correspondente ao grau de liberdade da carga em torno do eixo Y <bodyname> </bodyname> nome do elo correspondente ao corpo principal do servo motor

Tabela B.4: Configuração do plugin State space load.

Descrição:	plugin para sensoriamento da temperatura e pressão atmosférica com ruído.
Arquivo:	libgazebo_ros_temperature.so
Configurações:	<Topic> </Topic> nome do tópico para o usuário obter dados sensoriais <TempOffset> </TempOffset> offset de erro para dados de temperatura ruidosos <TempStandardDeviation> </TempStandardDeviation> desvio padrão de erro para dados de temperatura ruidosos <BaroOffset> </BaroOffset> offset de erro para dados de pressão ruidosos <BaroStandardDeviation> </BaroStandardDeviation> desvio padrão de erro para dados de pressão ruidosos <maxtemp> </maxtemp> valor máximo de temperatura <mintemp> </mintemp> valor mínimo de temperatura <maxbaro> </maxbaro> valor máximo de pressão <minbaro> </minbaro> valor mínimo de pressão <Nbits> </Nbits> quantidade de bits utilizados na digitalização

Tabela B.5: Configuração do plugin Temperature.

Descrição:	plugin para sensoriamento de todos os dados que o Gazebo disponibiliza de uma junta. (ângulo, velocidade angular e Torque)
Arquivo:	libgazebo_ros_universaljoint.so
Configurações:	<NameOfTopic> </NameOfTopic> nome do tópico para o usuário obter dados sensoriais <NameOfJoint> </NameOfJoint> nome da junta para sensoriamento <Axis> </Axis> Eixo de rotação da junta (”axis” para primeira junta e ”axis2” para segunda junta - gazebo contém juntas q permite dois graus de liberdade)

Tabela B.6: Configuração do plugin UniversalJointSensor.

Descrição:	plugin para sensoriamento de todos os dados que o Gazebo disponibiliza de um elo.
Arquivo:	libgazebo_ros_universallink.so
Configurações:	<NameOfTopic> </NameOfTopic> nome do tópico para o usuário obter dados sensoriais <NameOfLink> </NameOfLink> nome da elo para sensoriamento

Tabela B.7: Configuração do plugin UniversalLinkSensor.

Ordem de informações:	pose relativa em x pose relativa em y pose relativa em z pose relativa em phi pose relativa em theta pose relativa em psi velocidade relativa em x velocidade relativa em y velocidade relativa em z aceleração linear relativa em x aceleração linear relativa em y aceleração linear relativa em z força relativa em x força relativa em y força relativa em z velocidade angular relativa em x velocidade angular relativa em y velocidade angular relativa em z aceleração angular relativa em x aceleração angular relativa em y aceleração angular relativa em z conjugado mecânico relativa em x conjugado mecânico relativa em y conjugado mecânico relativa em z pose global em x pose global em y pose global em z pose global em phi pose global em theta pose global em psi velocidade global em x velocidade global em y velocidade global em z aceleração linear global em x aceleração linear global em y aceleração linear global em z força global em x força global em y força global em z velocidade angular global em x velocidade angular global em y velocidade angular global em z aceleração angular global em x aceleração angular global em y aceleração angular global em z conjugado mecânico global em x conjugado mecânico global em y conjugado mecânico global em z velocidade linear do centro de gravidade global em x velocidade linear do centro de gravidade global em y velocidade linear do centro de gravidade global em z pose linear do centro de gravidade global em x pose linear do centro de gravidade global em y pose linear do centro de gravidade global em z

Tabela B.8: Ordem de dados do plugin UniversalLinkSensor.

Descrição:	plugin para transmitir dados do sensor GPS para ROS.
Arquivo:	libgazebo_ros_gps.so
Configurações:	<gazebotopic> </gazebotopic> nome do tópico do gazebo onde GPS publica dados <rostopic> </rostopic> tópico do ROS <link> </link> nome do que o GPS está acoplado

Tabela B.9: Configuração do plugin modelo para transmissão de dados do GPS dos tópicos do Gazebo para tópicos do ROS.

Descrição:	plugin para transmitir dados do sensor IMU para ROS
Arquivo:	libgazebo_imu_gps.so
Configurações:	<gazebotopic> </gazebotopic> nome do tópico do gazebo onde IMU publica dados <rostopic> </rostopic> tópico do ROS <link> </link> nome do elo que o IMU está acoplado

Tabela B.10: Configuração do plugin modelo para transmissão de dados do IMU dos tópicos do Gazebo para tópicos do ROS.

Descrição:	plugin para transmitir dados do sensor sonar para ROS.
Arquivo:	libgazebo_ros_sonar.so
Configurações:	<gazebotopic> </gazebotopic> nome do tópico do gazebo onde sonar publica dados <rostopic> </rostopic> tópico do ROS <link> </link> nome do elo que o sonar está acoplado

Tabela B.11: Configuração do plugin modelo para transmissão de dados do Sonar dos tópicos do Gazebo para tópicos do ROS.

Descrição:	plugin para transmitir dados do sensor magnetômetro para ROS.
Arquivo:	libgazebo_ros_magnetometro.so
Configurações:	<gazebotopic> </gazebotopic> nome do tópico do gazebo onde magnetometro publica dados <rostopic> </rostopic> magnetometro está acoplado <link> </link> nome do elo que o magnetômetro está acoplado

Tabela B.12: Configuração do plugin modelo para transmissão de dados do Magnetômetro dos tópicos do Gazebo para tópicos do ROS.

Apêndice C

CMakeLists.txt

Esta seção foi extraída da página http://wiki.ros.org/catkin/CMakeLists.txt no dia 25/08/2017, visite-a para mais informações.

C.1 Visão geral e estrutura do arquivo CMakeLists.txt

O arquivo CMakeLists.txt armazena comandos de compilação e instalação de pacotes de software. Necessariamente, o arquivo **deve seguir o formato e a ordem a seguir**.

1. Versão CMake necessária (cmake_minimum_required)
2. Nome do pacote (project())
3. Encontrar outros pacotes CMake/Catkin necessários para compilação (find_package())
4. Habilitação de suporte para módulos Python (catkin_python_setup())
5. Geradores de Mensagens/Serviços/Ações do ROS (add_message_files(), add_service_files(), add_action_files())
6. Invocar geração de Mensagem/Serviço/Ação (generate_messages())
7. Especificar pacote de compilação, informação e exportação (catkin_package())
8. Bibliotecas/Executáveis para compilação (add_library()/add_executable()/target_link_libraries())
9. Testes de construção (catkin_add_gtest())
10. Regras de instalação (install())

C.2 Versão CMake

Todo arquivo CMakeLists.txt deve começar com a declaração da versão do sistema. A versão requisitada é a 2.8.3 ou superior.

```
cmake_minimum_required(VERSION 2.8.3)
```

C.3 Nome do Pacote

O próximo item corresponde ao o nome do pacote do ROS. No exemplo a seguir, o pacote é chamado *robot_brain*.

```
project(robot_brain)
```

Obs.: Após esse comando, é possível fazer referência do nome do projeto em qualquer outro lugar através do uso da variável $PROJECT_NAME.

C.4 Encontrando dependências de pacotes CMake

É necessário especificar quais outros pacotes precisam ser localizados para compilar o projeto. Essa especificação é realizada com o comando find_package e sempre há ao menos uma dependência pelo pacote catkin:

```
find_package(catkin REQUIRED)
```

Se o projeto depende de outros pacotes, eles são convertidos automaticamente em componentes do sistema catkin. Em vez de usar o comando find_package naqueles pacotes, é possível especificá-los como componentes para melhorar a legibilidade do script. O exemplo a seguir utiliza pacote 'nodelet" como componente.

```
find_package(catkin REQUIRED COMPONENTS nodelet)
```

Obs: É necessário usar somente o comando find_package para encontrar componentes. Não se deve adicionar dependência de tempo de execução.

C.4.1 O find_package()

Se um pacote é localizado através do comando find_package, então resultará na criação de várias variáveis de ambiente do script CMakeLists que fornecem informação sobre o pacote encontrado. As variáveis de ambiente descrevem onde os arquivos dos pacotes exportados estão, quais bibliotecas o pacote depende e os caminhos destas bibliotecas. Os nomes seguem a convenção <PACKAGE NAME>_<PROPERTY>:

<NAME>_FOUND - configurado como verdadeiro caso a biblioteca é encontrada, caso contrário, falso
<NAME>_INCLUDE_DIRS ou <NAME>_INCLUDES - Caminhos de inclusão exportados pelo pacote
<NAME>_LIBRARIES ou <NAME>_LIBS - Bibliotecas exportadas pelo pacote
<NAME>_DEFINITIONS - Definições exportadas pelo pacote

C.4.2 Por que os pacotes são especificados como componentes?

Pacotes não são realmente componentes catkin. Em vez disso, a característica de componentes foi utilizada no desenvolvimento do script para economizar tempo de digitação significativo.

Para pacotes, será vantajoso utilizar o comando find_package para encontrá-los como componentes. Eles estão em um conjunto de variáveis criado com o prefixo catkin_. Por exemplo, quando você estiver usando um pacote denominado "nodelet" no seu código, sugere-se utilizar:

```
find_package(catkin REQUIRED COMPONENTS nodelet)
```

Isto significa que os caminhos incluídos, bibliotecas, etc. exportados pelo pacote "nodelet" são também anexadas às variáveis catkin_. Por exemplo, catkin_INCLUDE_DIRS contém os caminhos de inclusão de não somente o pacote catkin mas também para para o pacote "nodelet".

Podemos de maneira alternativa achar o pacote "nodelet" com o comando:

```
find_package(nodelet)
```

Isto significa que os caminhos, bibliotecas e demais características do pacote "nodelet" não seriam adicionados às variáveis catkin_ . O que resulta em nodelet_INCLUDE_DIRS, nodelet_LIBRARIES, e etc.

As mesmas variáveis também são criadas usando

```
find_package(catkin REQUIRED COMPONENTS nodelet)
```

C.4.3 Boost

Caso esteja usando C++ e Boost, você necessite invocar o comando find_package() para Boost e especificar quais aspectos da Boost que você está usando como componentes. Boost é um conjunto de bibliotecas para a linguagem de programação C ++ que oferece suporte para tarefas e estruturas, como álgebra linear, geração de números pseudorandom, multithreading, processamento de imagem, expressões regulares e teste de unidade. Por exemplo, se você quiser usar threads da biblioteca Boost, você utilizaria o seguinte comando:

```
find_package(Boost REQUIRED COMPONENTS thread)
```

C.5 catkin_package()

O comando catkin_package() especifica informações do sistema catkin para o compilador.

Esta função deve ser utilizada antes de declarar quaisquer alvos com comandos add_library() ou add_executable(). A função tem 5 argumentos opcionais:

INCLUDE_DIRS - Caminhos de inclusão exportados pelo pacote
LIBRARIES - Bibliotecas exportadas do projeto
CATKIN_DEPENDS - Outros projetos catkin que este projeto depende
DEPENDS - Projetos não-catkin que este projeto depende
CFG_EXTRAS - Opções de configuração adicional

Observe o exemplo:

```
catkin_package(INCLUDE_DIRS include
LIBRARIES \${PROJECT_NAME}
CATKIN_DEPENDS roscpp nodelet
DEPENDS eigen opencv)
```

Isto indica que o diretório "include" dentro do diretório do pacote é o local que os cabeçalhos serão direcionados. A variável de ambiente $PROJECT_NAME avalia informação passada para a função project(), neste caso será "robot_brain". "roscpp" e "nodelet" são pacotes que necessitam estar presentes durante a compilação/execução deste pacote, já "eigen" e "opencv" são dependências do sistema que necessitam estar presentes para compilação/execução deste pacote.

C.6 Especificando alvos de compilação

Construir alvos pode ser realizadas de diversas formas, mas normalmente representam um das duas possibilidades:

Executable Target - programas a serem executados
Library Target - bibliotecas que podem ser usadas por alvos executáveis durante a compilação e/ou tempo de execução

C.6.1 Nomeando alvos

É muito importante notar que os nomes dos alvos de compilação no sistema catkin devem ser únicos sem considerar os diretórios em que eles foram compilados/instalados. Este é um requisito do CMake. Entretanto, nomes únicos são apenas necessários internamente para CMake. Alguém pode obter um alvo renomeado como outro nome usando o comando set_target_properties():

Exemplo:

```
set_target_properties(rviz_image_view
PROPERTIES OUTPUT_NAME image_view
PREFIX "")
```

Isto irá trocar o nome do alvo rviz_image_view para image_view na compilação e instalação de saídas.

C.6.2 Diretório customizado de saída

Enquanto o diretório de saída padrão para executáveis e bibliotecas é comum a um valor razoável, ele deve ser personalizado em certos casos. Isto é, uma biblioteca contendo ligações de Python deve ser realocada em um diretório diferente a fim de ser importável no Python.:

Exemplo:

```
set_target_properties(python_module_library
PROPERTIES LIBRARY_OUTPUT_DIRECTORY ${CATKIN_DEVEL_PREFIX}${CATKIN_PACKAGE_PYTHON_DESTINATION})
```

C.6.3 Caminhos de inclusão e caminhos de bibliotecas

Antes de especificar alvos, você precisa especificar onde os recursos como arquivos de cabeçalho e bibliotecas podem ser localizados:

Caminhos de inclusão
Caminhos de biblioteca
include_directories(<dir1>, <dir2>, ..., <dirN>)
link_directories(<dir1>, <dir2>, ..., <dirN>)

a) include_directories()

O argumento para o comando include_directories deve ser as variáveis _INCLUDE_DIRS geradas pelo chamada do comando find_package e qualquer diretório adicional que necessita ser incluído. Se você estiver usando o sistema catkin e Boost, a chamada de include_directories() deve ser parecer:

```
include_directories(include ${Boost_INCLUDE_DIRS} ${catkin_INCLUDE_DIRS})
```

O primeiro argumento "include" indica que o diretório include dentro do pacote faz parte também da caminho.

b) link_directories()

O comando link_directories() pode ser usada para adicionar novos caminhos de bibliotecas, no entanto, isto não é recomendado. Todos os pacotes catkin e CMake automaticamente tem sua informação de ligação adicionado quando são encontrados pelo find_package. Basta ligar as bibliotecas no target_link_libraries()
Exemplo:

```
link_directories(~/my_libs)
```

C.6.4 Alvos executáveis

Para especificar um alvo executável que deve ser construído, deve-se usar o comando CMake add_executable().

```
add_executable(myProgram src/main.cpp src/some_file.cpp src/another_file.cpp)
```

Isto irá construir um alvo executável denominado "myProgram"que, por sua vez, é constituídos de 3 arquivos fonte: src/main.cpp, src/some_file.cpp e src/another_file.cpp.

C.6.5 Alvos de bibliotecas

O comando CMake add_library() é usado para especificar bibliotecas para serem construídas. Por padrão o sistema catkin constrói bibliotecas dinâmicas.

```
add_library(${PROJECT_NAME} ${${PROJECT_NAME}_SRCS})
```

C.6.6 target_link_libraries

Use o comando target_link_libraries() para especificar quais bibliotecas um alvo executável é ligado. Isto é feito tipicamente depois da chamada add_executable(). Adicione ${catkin_LIBRARIES} se o ROS não for encontrado.
Sintaxe:

```
target_link_libraries(<executableTargetName>, <lib1>, <lib2>, ... <libN>)
```

Exemplo:

```
add_executable(foo src/foo.cpp)
add_library(moo src/moo.cpp)
target_link_libraries(foo moo)  -- This links foo against libmoo.so
```

Observe que não há necessidade para uso de link_directories() na maioria dos casos, pois a informação é automaticamente enviada via find_package().

C.7 Mensagens alvos, Serviços alvos e Ações alvos

Arquivos de mensagens (.msg), serviços (.srv), e ações (.action) no ROS requerem um passo de construção especial antes de serem construídas e usadas por pacotes do ROS. O objetivo destas macros é gerar arquivos de linguagem específica de programação de maneira que alguém utilize mensagens, serviços e ações na linguagem de programação desejada. O sistema de compilação gerará ligações usando todos os geradores disponíveis (ex. gencpp, genpy, genlisp, etc).

Estas são três macros fornecidos para tratar de mensagens, serviços e ações respectivamente:

```
add_message_files
add_service_files
add_action_files
```

Estas macros devem ser seguidas por uma chamada que invoca a geração:

```
generate_messages()
```

Restrições/Pré-requisitos importantes

Estas macros devem vir antes do comando catkin_package() para correta geração.

```
find_package(catkin REQUIRED COMPONENTS ...)
add_message_files(...)
add_service_files(...)
add_action_files(...)
generate_messages(...)
catkin_package(...)
...
```

O comando catkin_package() deve ter dependência (CATKIN_DEPENDS) do pacote message_runtime.

```
catkin_package(
...
CATKIN_DEPENDS message_runtime ...
...)
```

Deve-se usar find_package() para o pacote message_generation, seja sozinho ou como um componente do sistema catkin:

```
find_package(catkin REQUIRED COMPONENTS message_generation)
```

O arquivo package.xml deve conter uma dependência de construção do message_generation e uma dependência de tempo de execução message_runtime. Isso não é necessário se as dependências forem utilizadas indiretamente de outros pacotes. Se tiver um alvo que, mesmo indiretamente, depende de algum outro alvo que precise de mensagens/serviços/ações a serem construídas, precisa-se adicionar uma dependência explícita no alvo catkin_EXPORTED_TARGETS, para que eles sejam construídos na ordem correta. Este caso aplica-se quase sempre, a menos que o pacote realmente não use nenhuma parte do ROS. Infelizmente, essa dependência não pode ser propagada automaticamente. (some_target correspondem ao nome do alvo definido por add _executable()):

```
add_dependencies(some_target ${catkin_EXPORTED_TARGETS})
```

Se existir um pacote que cria mensagens e/ou serviços, bem como executáveis que usam estes, precisa-se criar uma dependência explícita no alvo da mensagem gerada automaticamente para que eles sejam construídos na ordem correta. (some_target correspondem ao nome do alvo definido por add_executable ()):

```
add_dependencies(some_target ${${PROJECT_NAME}_EXPORTED_TARGETS})
```

Se o pacote satisfizer as duas condições acima, precisa-se adicionar ambas as dependências, ou seja:

```
add_dependencies(some_target ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
```

C.7.1 Exemplo

Suponha que o pacote tenha duas mensagens em um diretório chamado "msg"chamadas "MyMessage1.msg"e "MyMessage2.msg"que dependem de std_msgs e sensor_msgs. Além disso, possui um serviço em um diretório chamado "srv"chamado "MyService.srv". O pacote define um executável que usa essas mensagens e serviços e um executável que usa algumas partes do ROS, mas não mensagens/serviços definidos neste pacote. Tal exemplo precisará do seguinte CMakeLists.txt:

```
# Obtém informação sobre as dependências de compilação do pacote
find_package(catkin REQUIRED
COMPONENTS message_generation std_msgs sensor_msgs)

# Declarar arquivos mensagens para serem construídos
add_message_files(FILES
MyMessage1.msg
MyMessage2.msg
)

# Declarar arquivos de serviços a serem construídos
add_service_files(FILES
MyService.srv
)

# Gerar as mensagens e serviços com as linguagems específica de programação
generate_messages(DEPENDENCIES std_msgs sensor_msgs)

# Declarar as dependências de tempo de execução do pacote
catkin_package(
CATKIN_DEPENDS message_runtime std_msgs sensor_msgs
)

# Define executáveis que usam mensagens.
add_executable(message_program src/main.cpp)
add_dependencies(message_program ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})

# Define executável que não utiliza quaisquer mensagens/serviços fornecidos pelo pacote
add_executable(does_not_use_local_messages_program src/main.cpp)
add_dependencies(does_not_use_local_messages_program ${catkin_EXPORTED_TARGETS})
```

Além disso, se for necessário criar ações e tenha um arquivo de especificação de ação chamado "MyAction.action"no diretório "ação", deve-se adicionar actionlib_msgs à lista de componentes que são encontrados com o sistema catkin e adicionar a seguinte chamada antes da chamada do comando generate_messages (...):

```
add_action_files(FILES
MyAction.action
)
```

Além disso, o pacote deve ter uma dependência de compilação em actionlib_msgs.

C.8 Habilitando suporte a módulos de Python

Se o pacote ROS fornecer alguns módulos Python, deve-se criar um arquivo setup.py e chamar

```
catkin_python_setup()
```

Antes da chamada para generate_messages() e catkin_package().

C.9 Testes unitários

Há uma macro específica de catkin para manipulação de testes unitários baseados em gtest chamado catkin_add_gtest ().

```
catkin_add_gtest(myUnitTest test/utest.cpp)
```

C.10 Passo opcional: Especificando alvos instaláveis

Após o tempo de compilação, os alvos são colocados no espaço de desenvolvimento do espaço de trabalho catkin. No entanto, muitas vezes queremos instalar alvos no sistema para que eles possam ser usados por outros ou para uma pasta local para testar uma instalação no nível do sistema. Em outras palavras, se deseja-se ser capaz de fazer uma "instalação"do código, você precisa especificar onde os objetivos devem acabar.

Este é realizado usando o comando CMake install() que possui como argumentos:

TARGETS - alvos para instalar
ARCHIVE DESTINATION - bibliotecas estáticas e DLL (Windows) .stub
LIBRARY DESTINATION - bibliotecas dinâmicas não DLL e módulos
RUNTIME DESTINATION - Alvos executáveis e DLL (Windows) com estilo de bibliotecas compartilhadas

Observe o exemplo:

```
install(TARGETS ${PROJECT_NAME}
ARCHIVE DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
LIBRARY DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
```

Além desses destinos padrão, alguns arquivos devem ser instalados em pastas especiais. Isto é, uma biblioteca contendo módulos Python deve ser instalada em uma pasta diferente para ser importável no Python:

```
install(TARGETS python_module_library
ARCHIVE DESTINATION ${CATKIN_PACKAGE_PYTHON_DESTINATION}
LIBRARY DESTINATION ${CATKIN_PACKAGE_PYTHON_DESTINATION}
```

C.10.1 Instalando scripts executáveis de Python

Para o código Python, a regra de instalação parece diferente, pois não há uso das funções add_library() e add_executable(), de modo que o CMake determina quais arquivos são alvos e que tipo de alvos eles são. Em vez disso, usar o seguinte comando no arquivo CMakeLists.txt:

```
catkin_install_python(PROGRAMS scripts/myscript
DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
```

Informações detalhadas sobre a instalação de scripts e módulos de python, bem como as melhores práticas para layout de pastas podem ser encontradas no manual catkin.

Se instala-se apenas scripts Python e não se fornece módulos, não é preciso criar o arquivo setup.py acima mencionado, nem chamar catkin_python_setup().

C.10.2 Instalando arquivos de cabeçalho

Os arquivos de cabeçalho também devem ser instalados na pasta ”incluir”, isso geralmente é feito instalando os arquivos de uma pasta inteira (opcionalmente filtrada por padrões de nomes de arquivos e excluindo subpastas SVN). Isso pode ser feito com uma regra de instalação que é a seguinte:

```
install(DIRECTORY include/${PROJECT_NAME}/
DESTINATION ${CATKIN_PACKAGE_INCLUDE_DESTINATION}
PATTERN ".svn" EXCLUDE
```

Ou se a subpasta localizada na pasta incluir não corresponde ao nome do pacote:

```
install(DIRECTORY include/
DESTINATION ${CATKIN_GLOBAL_INCLUDE_DESTINATION}
PATTERN ".svn" EXCLUDE
```

Instalando arquivos roslaunch Files ou outros recursos

Outros recursos são arquivos launch que podem ser instalados em $CATKIN_PACKAGE_SHARE_DESTINATION:

```
install(DIRECTORY launch/
DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}/launch
PATTERN ".svn" EXCLUDE)
```

Apêndice D

package.xml

Esta seção possui conteúdo retirado da página http://wiki.ros.org/catkin/package.xml no dia 25/08/2017. Para mais detalhes, visite-a.

D.1 Visão geral

O manifesto do pacote é um arquivo XML chamado package.xml que deve ser incluído no diretório raíz de qualquer pacote compatível com catkin. Este arquivo define propriedades sobre o pacote, como o nome do pacote, números de versão, autores, responsáveis e dependências em outros pacotes catkin.

As dependências do pacote do sistema são declaradas em package.xml. Se eles estão faltando ou incorretos, é possível criar a partir da fonte e executar testes em sua própria máquina, mas o pacote não funcionará corretamente quando for lançado para a comunidade ROS. Outros dependem dessas informações para instalar o software necessário para usar seu pacote.

D.2 Formato 2 (Recomendado)

Este é o formato recomendado para novos pacotes. Também é recomendado que os pacotes anteriores do formato 1 sejam migrados para o formato 2. Para obter instruções sobre como migrar do formato 1 para o formato 2, consulte a página http://docs.ros.org/indigo/api/catkin/html/howto/format2/migrating_from_format_1.html#migrating-from-format1-to-format2 para mais informações.

A documentação completa para o formato 2 pode ser encontrada em http://www.ros.org/reps/rep-0140.html.

D.2.1 Estrutura básica

Cada arquivo package.xml tem uma tag <package> como tag raíz do documento.

```
<package format="2">

</package>
```

D.2.2 Tags requisitadas

Há um conjunto mínimo de tags que necessitam de ser alocadas dentro da tag <package> para tornar o manifesto do pacote completo.

<name> - O nome do pacote
<version> - O número da versão do pacote (obrigatório ter 3 inteiros separados por pontos)
<description> - Descrição do conteúdo do pacote
<maintainer> - Nome das pessoas que realizam a manutençaõ do pacote
<license> - A(s) licença(s) de software (por exemplo, GPL, BSD, ASL) sob a qual o código é liberado.

Observe um exemplo de manifesto para um pacote fictício denominado foo_core.

```
<package format="2">
        <name>foo_core</name>
        <version>1.2.4</version>
        <description>
                This package provides foo capability.
        </description>
        <maintainer email="ivana@osrf.org">Ivana Bildbotz</maintainer>
        <license>BSD</license>
</package>
```

D.2.3 Dependências

O manifesto do pacote com tags mínimas não especifica nenhuma dependência em outros pacotes. Os pacotes podem ter seis tipos de dependências:

- **Dependências de compilação** especifica quais pacotes são necessários para compilar este pacote. Este é o caso quando algum arquivo destes pacotes é requisitado em tempo de compilação. Isto pode ser realizado incluindo cabeçalhos destes pacotes em tempo de compilação, ligando bibliotecas destes pacotes ou requisitando algum outro recurso em tempo de compilação (especialmente quando estes pacotes são encontrados por find_package() no CMake). Num cenário de cross-compilação construir dependências são próprias para uma arquitetura segmentada.
- **Dependências de exportação** especifica quais pacotes são necessários para criar bibliotecas. Este é o caso quando você inclui indiretamente seus cabeçalhos em cabeçalhos públicos neste pacote (especialmente quando estes pacotes são declarados como (CATKIN_)DEPENDs em catkin_package() no CMake).
- **Dependências de execução** Especifica quais pacotes são necessários para executar o código neste pacote. Este é o caso quando você depende de bibliotecas compartilhadas neste pacote (especialmente quando esses pacotes são declarados como (CATKIN_)DEPENDs em catkin_package() no CMake).
- **Dependências de testes** Especifica apenas dependências adicionais para testes unitários. Nunca devem duplicar quaisquer dependências já mencionadas como dependências de compilação ou execução.
- **Dependências de ferramentas de construção** Especifica as ferramentas do sistema de compilação que este pacote precisa construir. Normalmente, a única ferramenta de compilação necessária é o sistema catkin. Em um cenário de compilação cruzada, as dependências da ferramenta de compilação são para a arquitetura na qual a compilação é executada.
- **Dependências de ferramentas de documentação** especifica ferramentas que esse pacote precisa para gerar documentação.

Esses seis tipos de dependências são especificados usando as seguintes tags respectivas:

<depend> especifica uma dependência de compilação, exportação e execução. Esta é a etiqueta de dependência mais usada.
<buildtool_depend>
<build_depend>
<build_export_depend>
<exec_depend>
<test_depend>
<doc_depend>

Todos os pacotes possuem pelo menos uma dependência. A seguir está um exemplo de uma dependência de ferramenta de compilação no sistema catkin.

```
<package>
        <name>foo_core</name>
        <version>1.2.4</version>
        <description>
                This package provides foo capability.
```

```
        </description>
        <maintainer email="ivana@osrf.org">Ivana Bildbotz</maintainer>
        <license>BSD</license>
        <buildtool_depend>catkin</buildtool_depend>
</package>
```

Um exemplo mais realista que especifica as dependências de compilação, exec, teste e doc pode parecer o seguinte.

```
<package>
        <name>foo_core</name>
        <version>1.2.4</version>
        <description>
                This package provides foo capability.
        </description>
        <maintainer email="ivana@willowgarage.com">Ivana Bildbotz</maintainer>
        <license>BSD</license>
        <url>http://ros.org/wiki/foo_core</url>
        <author>Ivana Bildbotz</author>
        <buildtool_depend>catkin</buildtool_depend>
        <depend>roscpp</depend>
        <depend>std_msgs</depend>
        <build_depend>message_generation</build_depend>
        <exec_depend>message_runtime</exec_depend>
        <exec_depend>rospy</exec_depend>
        <test_depend>python-mock</test_depend>
        <doc_depend>doxygen</doc_depend>
</package>
```

D.2.4 Metapackages

Muitas vezes, é conveniente agrupar vários pacotes como um único pacote lógico. Isso pode ser obtido através de metapackages. Um metapackage é um pacote normal com a seguinte *tag* de exportação no package.xml:

```
<export>
        <metapackage />
</export>
```

Além de uma dependência <buildtool_depends> necessária, os metapackages só podem ter dependências executadas em pacotes dos quais eles agrupam.

Além disso, um metapackage possui um arquivo CMakeLists.txt obrigatório:

```
cmake_minimum_required(VERSION 2.8.3)
project(<PACKAGE_NAME>)
find_package(catkin REQUIRED)
```

Note: substitua <PACKAGE_NAME> com o nome do metapackage.

D.2.5 Tags adicionais

<url> - Um URL para obter informações sobre o pacote, normalmente uma página do wiki no ros.org.
<author> - O(s) autores do pacote

D.3 Formato 1 (legado)

Os pacotes catkin mais antigos usam o formato 1. Se a tag <package> não tiver um atributo de formato, é um pacote de formato 1. Use o formato 2 para novos pacotes.

D.3.1 Estrutura básica

Cada arquivo package.xml tem a tag <package> como tag raís no documento.

```
<package>

</package>
```

D.3.2 Tags Necessárias

Há um conjunto mínimo de tags que precisam ser aninhadas dentro da tag <package> para tornar o pacote manifesto completo.

<name> - O nome do pacote
<version> - A versão do pacote (Obrigatório ter 3 inteiros separados por pontos)
<description> - Uma descrição do conteúdo do pacote
<maintainer> - O nome da pessoa responsáveis pela manutenção do pacote
<license> - A(s) licença(s) de software (por exemplo, GPL, BSD, ASL) sob a qual o código é liberado.

Como exemplo, aqui está o pacote manifesto para um pacote de ficção chamado foo_core.

```
<package>
        <name>foo_core</name>
        <version>1.2.4</version>
        <description>
                This package provides foo capability.
        </description>
        <maintainer email="ivana@willowgarage.com">Ivana Bildbotz</maintainer>
        <license>BSD</license>
</package>
```

D.3.3 Dependências de compilação, execução e testes

O manifesto do pacote com tags mínimas não especifica nenhuma dependência em outros pacotes. Pacotes podem ter quatro tipos de dependências:

- **Dependências de ferramentas de construção** especifica as ferramentas do sistema de compilação que este pacote precisa construir. Normalmente, a única ferramenta de compilação necessária é o sistema catkin. Em um cenário de compilação cruzada, as dependências da ferramenta de compilação são para a arquitetura na qual a compilação é executada.
- **Dependências de construção** especifica quais pacotes são necessários para criar este pacote. Este é o caso quando um arquivo desses pacotes é necessário no momento da compilação. Isso pode incluir cabeçalhos desses pacotes no momento da compilação, vinculando as bibliotecas desses pacotes ou exigindo qualquer outro recurso em tempo de compilação (especialmente quando esses pacotes são find_package()-ed no CMake). Em um cenário de compilação cruzada, as dependências de compilação são para a arquitetura segmentada.
- **Dependências de execução** especifica quais pacotes são necessários para executar o código neste pacote ou criar bibliotecas neste pacote. Este é o caso quando você depende de bibliotecas compartilhadas ou inclui transitivamente seus cabeçalhos em cabeçalhos públicos (especialmente quando estes pacotes são declarados como (CATKIN_)DEPENDs em catkin_package() no CMake).
- **Dependências de teste** especifica apenas dependências adicionais para testes unitários. Eles nunca devem duplicar quaisquer dependências já mencionadas como dependências de compilação ou execução.

Esses quatro tipos de dependências são especificados usando as seguintes tags:

- <buildtool_depend>
- <build_depend>
- <run_depend>

- <test_depend>

Todos os pacotes possuem pelo menos uma dependência de ferramenta de compilação no catkin conforme mostra o exemplo a seguir.

```
<package>
        <name>foo_core</name>
        <version>1.2.4</version>
        <description>
                This package provides foo capability.
        </description>
        <maintainer email="ivana@willowgarage.com">Ivana         Bildbotz</maintainer>
        <license>BSD</license>
        <buildtool_depend>catkin</buildtool_depend>
</package>
```

Um exemplo mais realista que especifica as dependências de compilação, tempo de execução e teste pode ser o seguinte.

```
<package>
        <name>foo_core</name>
        <version>1.2.4</version>
        <description>
                This package provides foo capability.
        </description>
        <maintainer email="ivana@willowgarage.com">Ivana Bildbotz</maintainer>
        <license>BSD</license>
        <url>http://ros.org/wiki/foo_core</url>
        <author>Ivana Bildbotz</author>
        <buildtool_depend>catkin</buildtool_depend>
        <build_depend>message_generation</build_depend>
        <build_depend>roscpp</build_depend>
        <build_depend>std_msgs</build_depend>
        <run_depend>message_runtime</run_depend>
        <run_depend>roscpp</run_depend>
        <run_depend>rospy</run_depend>
        <run_depend>std_msgs</run_depend>
        <test_depend>python-mock</test_depend>
</package>
```

D.3.4 Metapackages

Muitas vezes, é conveniente agrupar vários pacotes como um único pacote lógico. Isso pode ser conseguido através de metapackages. Um metapackage é um pacote normal com a seguinte tag de exportação no package.xml:

```
<export>
        <metapackage />
</export>
```

Além de uma dependência <buildtool_depends> necessária, os metapackages só podem ter dependências executadas em pacotes dos quais eles agrupam.

Além disso, um metapackage possui um arquivo CMakeLists.txt obrigatório, exigido:

```
cmake_minimum_required(VERSION 2.8.3)
project(<PACKAGE_NAME>)
find_package(catkin REQUIRED)
catkin_metapackage()
```

Note: substitua <PACKAGE_NAME> com o nome do metapackage.

D.3.5 Tags Adicionais

- <url> - Um URL para obter informações sobre o pacote, normalmente uma página do wiki no ros.org.
- <author> - O(s) autor(es) do pacote

Apêndice E

Tutorial sobre como executar uma simulação via Hardware-in-the-loop

O processo para a execução de uma simulação via hardware-in-the-loop é simples, e consiste básicamente em três passos:

- Configuração do ProVANT Simulator
- Configuração do Hardware Embarcado
- Inicialização da simulação

Este apêndice instrui sobre o procedimento para a execução de uma simulação via Hardware-in-the-loop. Inicialmente, é demonstrado como configurar o Provant Simulator e, em seguida, é descrito o processo de compilação e *upload* de um código embarcado de um controlador exemplo já disponível no github do projeto ProVANT para a placa STM32f4 discovery.

E.1 Configurando ProVANT simulator

Inicialmente, o usuário deve inicializar o ProVANT Simulator (Dúvidas sobre o processo leia o capítulo 2) e, logo após, abrir o o mundo *vant2.world*. Este cenário já possui as configurações necessárias para a simulação HIL com o respectivo código exemplo.

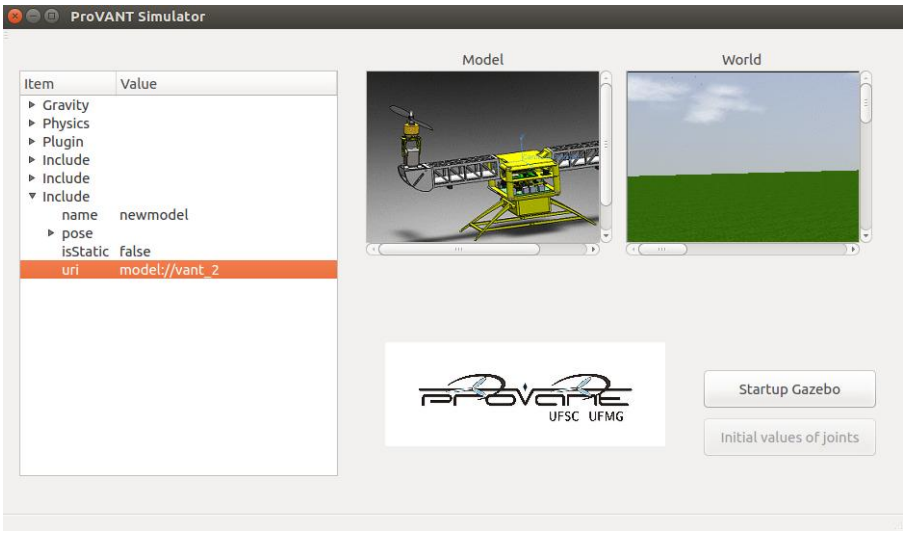

Figura E.1: Tela principal

Através da interface gráfica, clicar duas vezes no nome do modelo então abrirá a tela de configurações na tela de configurações de controlador. Nesta tela selecione a caixa de seleção "*Hardware-in-the-loop*". Pressione *OK* e volte para a tela inicial.

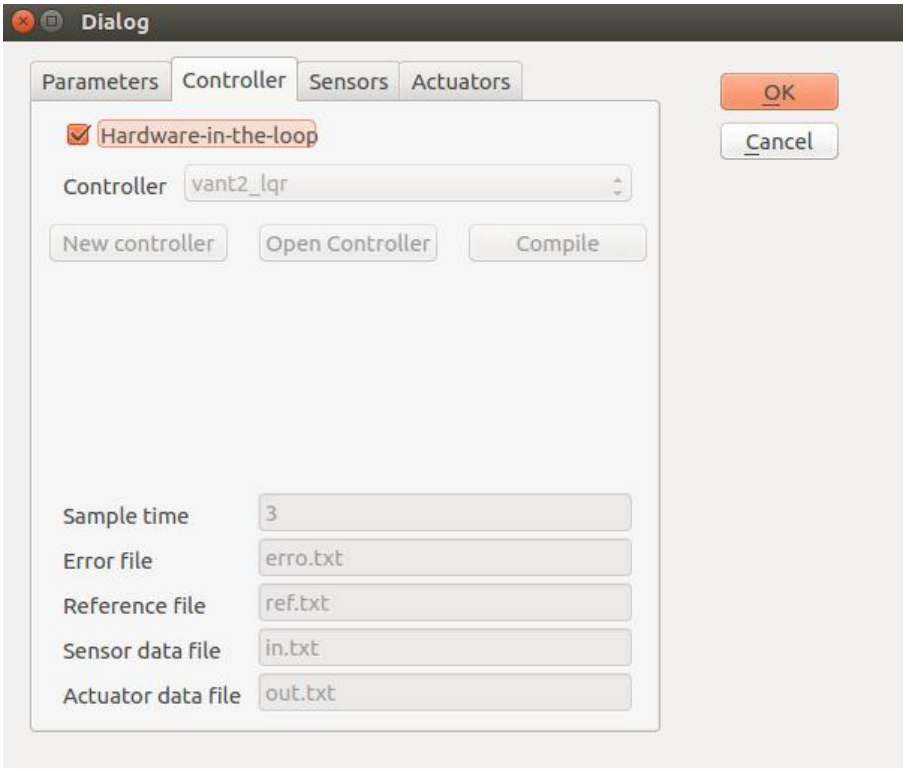

Figura E.2: Seleção de simulação via Hardware-in-the-loop

E.2 Configurando hardware embarcado

Acesse ao github privado do projeto de software embarcado do ProVANT (caso não possua credencial, solicite-a ao dono do repositório sua permissão), entre na branch *hil-base*. Abra o projeto *hil-base* no eclipse e compile-o, utilizando a opção *build* (Detalhes sobre a arquitetura de software embarcado então no próprio github).

Em seguida, ligue a placa STM32f4 discovery na entrada USB do computador (para futuro upload de código para a placa) e conecte a saída UART2, utilizando uma FTDI, numa outra porta serial para o recebimento de sinais de sensores e atuadores.

E.3 Incializando simulação

Faça upload do código via opções *Debug* ou *Run* do Eclipse. Caso for utiizado a opção *Debug* é necessário que usuário saia do breakpoint inicial por padrão do Eclipse. Tendo realizado os todos os passos anteriores a simulação funcionará corretamente.

Referências Bibliográficas

Alfaro, R. (2016). Predictive Control Strategies for Unmanned Aerial Vehicles in Cargo Transportation Tasks. Master's thesis, Universidade federal de Santa Catarina.

Cardoso, D. N. (2016). Adaptative Control Strategies For Improved Forward Flight of a Tilt-Rotor UAV. Master's thesis, Universidade Federal de Minas Gerais.

de Almeida Neto, M. M. (2014). Control Strategies of a Tilt-rotor UAV for Load Transportation. Master's thesis, Universidade Federal de Minas Gerais.

Donadel, R. (2015). Modeling and Control of a Tiltrotor Unmanned Aerial Vehicle for Path Tracking. Master's thesis, Universidade Federal de Santa Catarina.

Lara, A. V., S.Rego, B., Raffo, G. V., & Arias-Garcia, J. (2017). Desenvolvimento de um ambiente de simulação de VANTs Tilt-rotor para testes de estratégias de controle. *Anais do XIII SBAI*.

Rego, B. S. (2016). Path Tracking Control of a Suspended Load Using a Tilt-Rotor UAV. Master's thesis, Universidade Federal de Minas Gerais.